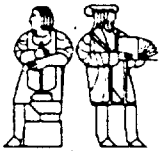


LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-485

CENTRAL-SERVER-BASED ORPHAN DETECTION FOR ARGUS

Steven C. Markowitz

May 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR 485	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Central-Server-Based Oprhan Detection for Argus

12. PERSONAL AUTHOR(S)
Steven C. Markowitz

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	May 1990	149

16. SUPPLEMENTARY NOTATION

17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	orphan dectection, distribution systems, highly-available, replication

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

One problem that can arise in a distributed computer system is that of *orphans*. These are computations that continue to execute even though their results are no longer needed. They can arise as a result of aborted transactions or node crashes. Orphans are bad because they waste resources and can see inconsistent data, causing a program to behave unpredictably. The Argus system makes use of an algorithm to detect and destroy orphans before they can cause harm. This scheme does not delay normal computations and avoids unnecessary communication between nodes. This is accomplished by piggybacking orphan detection information on normal system messages. The current version of this algorithm is impractical because of the large amount of information that must be included in all messages between nodes.

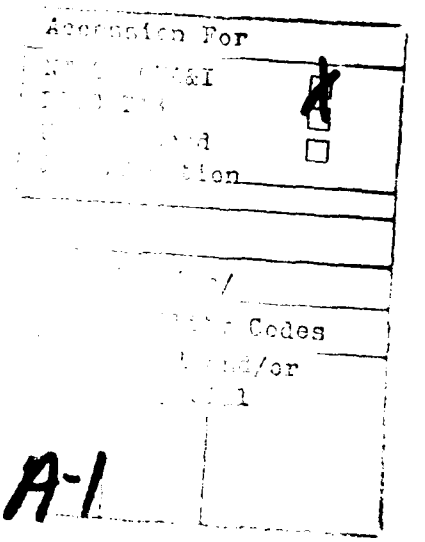
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

Central-Server-Based Orphan Detection for Argus

by

Steven C. Markowitz

May 1990



This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contracts N00014-83-K-0125 and N00014-89-J-1988.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

Central-Server-Based Orphan Detection for Argus

by

Steven C. Markowitz

Submitted to the Department of
Electrical Engineering and Computer Science
on May 18, 1990 in partial fulfillment of the requirements
for the degree of Master of Science

Abstract

One problem that can arise in a distributed computer system is that of *orphans*. These are computations that continue to execute even though their results are no longer needed. They can arise as a result of aborted transactions or node crashes. Orphans are bad because they waste resources and can see inconsistent data, causing a program to behave unpredictably. The Argus system makes use of an algorithm to detect and destroy orphans before they can cause harm. This scheme does not delay normal computations and avoids unnecessary communication between nodes. This is accomplished by piggybacking orphan detection information on normal system messages. The current version of this algorithm is impractical because of the large amount of information that must be included in all messages between nodes.

For this thesis, an optimization of the Argus orphan detection algorithm was implemented. The optimization reduces the cost of orphan detection by storing orphan detection information in a central server. This reduces the amount of orphan information sent in normal system messages. The performance of the central server optimization was examined under selected system loads and compared to that of another optimization called *deadlining*. These results show that while the central server method is more efficient than the unoptimized version of the algorithm, it remains costly and the deadlining optimization is superior.

Thesis Supervisor: Barbara H. Liskov
Title: NEC Professor of Software Science and Engineering

Keywords: orphan detection, distributed systems, highly-available, replication

Acknowledgements

I would like to thank Barbara Liskov, my thesis advisor, for suggesting this thesis topic, for making many valuable suggestions during the course of this project, and for reading the drafts of this thesis promptly. I would also like to thank Dorothy Curtis and Paul Johnson for answering my many questions about the Argus system. Finally, I would like to thank Thu Nguyen for taking the time to discuss his work on orphan detection with me, and for explaining the structure and use of the load generation and measurement tools that he developed.

Contents

List of Figures

Chapter 1

Introduction

One problem that can arise in a distributed computer system is *orphaned* computations, or *orphans* for short. Orphans are computations whose results are no longer needed. They can arise in two different ways. An orphan can be created when a computation running at one node of the system makes a request at another node and then aborts while the request is still in progress. Any results returned by the request cannot be used, because the request's parent no longer exists. An orphan created by the abort of its parent, or of some other ancestor, is called an *abort orphan*. An orphan can also be created when a computation performs a request at a node that later crashes, after the request has completed, but while the parent (or some other ancestor) is still active. If the computation modified data at the remote node, the modifications could have been lost in the crash. If the parent acquired locks on data at the remote node, the locks may have been lost in the crash, allowing another computation to modify data upon which the parent relies. In either case, any results produced by the parent may be invalid; for this reason the parent is considered an orphan. An orphan created in this way is called a *crash orphan*. Any active descendants of a crash orphan are also considered crash orphans.

Orphans are bad for several reasons: they waste resources, they can damage data, and they can see inconsistent data, causing a program to behave unpredictably. Several methods [3, 10, 11] have been developed to detect orphans and to destroy them before they cause harm.

This thesis considers orphan detection in Argus [5], an experimental system for distributed programming. An algorithm for identifying orphans and destroying them before they cause harm has already been included in the implementation of Argus [9]. This algorithm involves adding information related to orphan detection to normal system messages; this is known as "piggybacking". The Argus orphan detection algorithm never destroys non-orphan computations, and always destroys orphans before they can see inconsistent data, i.e. data that a non-orphan could not possibly see.

The algorithm used in Argus is too inefficient to be used in a production system, because of the large amount of orphan information that must be included in each message. Two optimizations have been proposed to deal with this problem. One of these, called *deadlining*, was proposed by Edward Walker [14] and implemented by Thu Nguyen [12]. It involves discarding orphan-related information after a certain period of time, when it is no longer needed, to reduce the amount of information included in each message. A second optimization, proposed in [6], involves storing orphan detection information in a central server, which is queried only when necessary.

This thesis presents an implementation of the central server optimization to orphan detection in Argus. We consider various issues in the design and implementation of this approach, discuss the details of our implementation, and present the results of experiments testing the performance of the algorithm.

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of the Argus programming language and run-time system. We explain the major features of Argus that are needed to understand the remainder of this paper. Chapter 3 describes the Argus orphan detection algorithm. We examine both the original, unoptimized form of the algorithm and the optimized version implemented as part of this thesis.

Chapters 4 and 5 describe the central server used to hold orphan information. Chapter 4 gives the specifications for the server and an overview of its implementation. In Chapter 5, the implementation of the server is discussed in greater detail.

Chapter 6 examines the modifications made to the Argus system to allow it to use our server for orphan detection. We first describe the structure of the Argus run-time system, and the way in which many common operations are performed. Next we examine the procedure used to detect and destroy crash orphans and abort orphans, and the implementation of system modules used for this purpose.

Chapter 7 examines the performance of the orphan detection algorithm under different system loads. We consider the efficiency of individual operations related to orphan detection and of the system as a whole. Our results are compared to those obtained by Nguyen [12] for the deadlining optimization of the Argus orphan detection algorithm. Finally, Chapter 8 presents our conclusions, discusses related work, and gives some suggestions for future research.

Chapter 2

Overview of Argus

Argus [5, 8] is a programming language and run-time system designed to support distributed programs. These programs run on independent machines, each with its own local state, that communicate with each other only through a communication network. The main facilities that Argus provides for distributed programming are long-lived resilient objects, called *guardians*, and atomic transactions, called *actions* for short.

2.1 Guardians

An Argus program consists of a number of modules called *guardians*. Guardians are the logical nodes of an Argus system. Each guardian runs at a single physical node of the system, but there may be many guardians at each node. A guardian is used to encapsulate data objects; other guardians can access these objects only through calls to special operations called *handlers*, which are provided by the guardian at which the data resides. Guardians use *remote procedure calls* [11] to invoke each other's handlers. The definition of a guardian also includes operations called *creators*, which are used to create new guardians of the same guardian type. In addition to creators and handlers, a guardian may have code to perform background tasks.

Guardians are resilient to crashes; each guardian maintains a stable state that can be restored from stable storage [4] during crash recovery. A guardian can also contain volatile objects, which are not saved on stable storage. Because volatile objects are not saved on

stable storage, their values are lost in the event of a crash. Therefore, they should only contain information that can be recovered to a correct state using the stable data. A guardian may include recovery code to be run when the guardian recovers from a crash, before it resumes processing handler calls. This code is responsible for reconstructing the volatile data from the stable state.

At each node in the system, Argus provides a special guardian called the *guardian manager*. The guardian manager is responsible for the creation of new guardians, and for restarting guardians after a crash.

In Argus, it is also possible to create modules called *programs*. Unlike guardians, programs do not have stable storage, and are not resilient to crashes. Programs can create guardians and make handler calls, but have no handlers themselves. Therefore, programs cannot receive calls. Programs are often used to create new guardians or to serve as the "front end" between guardians and users.

2.2 Actions and Atomic Objects

Argus allows computations to run as atomic transactions, or *actions* for short. Each action has the property that it either runs to completion or has no effect. In addition, actions are serializable, i.e. the effect of a set of concurrent actions is equivalent to the some serial execution of them. In Argus, actions can be nested. An action may have subactions, which can commit or abort independently from the parent action. Many subactions of the same parent can run concurrently. These subactions are serializable with each other. Subactions do not run concurrently with their parents; execution of the parent is suspended until the subaction completes. Each Argus computation starts as a top-level action (*topaction*). Each topaction runs at the guardian where it was created. When an action makes handler calls, the calls are run as subactions of the caller. Each handler action runs at the guardian to which the handler belongs, and has direct access to that guardian's local objects. A handler action may in turn invoke handlers at other guardians, creating new subactions of its own.

Argus provides special *atomic objects* that are used for synchronization and recovery. Synchronization is provided by granting read and write locks to actions that access atomic objects. An action may acquire a read lock for an object if no other action holds a write lock for it; a write lock may be acquired only if no other actions hold read or write locks for the object. There is one exception to this rule: an action may acquire a lock for an object if the only actions holding conflicting locks are its ancestors. Atomic objects provide recovery by means of versions. When an atomic object is modified by an action, a new version of the object is created, and the changes are made only to the new version. This allows the old version to be restored if the action aborts.

When an action aborts, any locks that it holds are released; the new versions of any objects modified by the action are discarded. The commit of an action causes its locks and versions to be inherited by its parent. In Argus, the commit of an action is relative: the effects of a committing action cannot immediately be seen by other actions; they are only visible to the committing action's parent and the parent's descendants. If the parent action commits, any changes will become visible to its own parent (i.e. the first action's grandparent); if the parent aborts, the effects of the child are completely undone. The effects of a committed topaction are permanent and are visible to all other actions. When an action has committed, and all of its ancestors up to and including its topaction ancestor have also committed, the action is said to have *committed through the top.*

The commit of a topaction causes the two-phase commit protocol [1] to be performed. This ensures that the action either commits everywhere or aborts everywhere. This is done only for topactions, not for subactions.

It is also possible to create *nested topactions.* A nested topaction runs as a child of the action that created it. However, the effects of a committed nested topaction are permanent, regardless of whether the action's parent commits or aborts. Two-phase commit is performed for nested topactions as well as for ordinary topactions. Nested topactions are not permitted to acquire locks held by their ancestors, or vice versa. Nested topactions can

be used to perform benevolent side effects that should not be undone even if the parent action aborts.

When a guardian crashes, all active local actions are aborted; their locks are released and their versions are discarded. In addition, if any local actions have committed, but have not yet committed through the top, their locks and versions are lost and their ancestors (and other relatives) may become orphans.

2.3 Mutexes

Argus provides another special type of object for synchronization. These objects are called *mutexes*, for *mut*ual *ex*clusion. Mutexes are similar to semaphores, except that in addition to providing a lock for synchronization, a mutex also contains data. An action obtains access to a mutex object by executing the *seize* statement; this gives the action a lock for the mutex. The mutex lock is held while the body of the *seize* statement is executed; it is then released. Only one action at a time can have a given mutex seized. When an action attempts to seize a mutex that is already seized, it must wait until the mutex is released, before it can proceed. Unlike atomic objects, mutexes do not have multiple versions; therefore, modifications made to mutex data are not undone automatically if the modifying action aborts. To ensure that a mutex is not left in an inconsistent state, an action cannot be aborted (except by the crash of its guardian) while it has a mutex seized. The Argus system does not copy a mutex to stable storage while any action has the mutex seized; this guarantees that the mutex is not left in an inconsistent state if the guardian crashes.

In the preceding discussion, many features of Argus that are not used in the remainder of this paper were omitted. For additional information about Argus, the reader is referred to [5, 8].

Chapter 3

Argus Orphan Detection Algoritm

The Argus orphan detection scheme is designed to achieve several goals: we wish to avoid unnecessary communication, delaying normal computations, and waiting for responses from other nodes. For example, if an action is about to abort, and has active descendants at other guardians, we do not want to wait until all of its these guardians have been notified and the action's descendants have been destroyed; inability to communicate with the other guardians may be the reason why the parent action is being aborted. Argus avoids this problem by including orphan detection information in ordinary messages ("piggybacking"). The method used guarantees that orphans are destroyed before they can see "inconsistent" data, i.e. data that a non-orphan could not possibly see.

3.1 Basic Algorithm

The orphan detection algorithm used in Argus is described in detail in [9] and [14]. It involves the use of two data structures at each guardian, called *done* and *map*, which are used to detect abort orphans and crash orphans, respectively.

A guardian's copy of *done* is a list of actions that the guardian knows have aborted. When a local action aborts, its name (called *aid*, for *a*ction *id*entifier) is added to this list. Any action having an ancestor in *done* must be an orphan and should be destroyed. To save space, an action's *aid* is removed from *done* when an ancestor's *aid* is added; any descendant of the first action must also be descended from the second, and will still be detected as an

orphan. In addition, an action's *aid* is not added to *done* unless the action has remote descendants. Without remote descendants, abort orphans cannot have been created; any local descendants are destroyed when the action aborts. An abort that causes an entry to be added to *done* is called a *crucial abort.*

A copy of *done* is included in (almost) all messages sent between guardians. (There are a few messages in which it is not actually necessary to include a copy of *done*; see [14]). Each message also contains the *aid* of the action (if any) on behalf of which the message was sent.

When a guardian receives a message containing a piggybacked *done* it first aborts any local actions descended from actions listed in the incoming *done*. (An action's *aid* contains the *aid*s of all of the action's ancestors, making it easy to tell how one action is related to another). Next, the incoming *done* is merged (unioned) with the guardian's own copy of *done*. Finally, the guardian determines whether the sender of the message is an orphan. If the sending action is descended from some action in the receiving guardian's *done*, the message is refused, and the sender's guardian is notified that the sending action is an orphan. Otherwise, the message is processed normally.

To deal with crash orphans, each guardian maintains a data structure called *map* — a list of other guardians and the number of times they have crashed. In general, the *map* will not be completely up to date, but will reflect all crashes that the guardian "knows about". If the *map* of a guardian G contains a crash count value of N for another guardian H, then either H has really crashed N times, or no information has propagated from H to G since the $N+1$st crash; in this case it is safe for G to behave as though H has crashed only N times. Each guardian records its own crash count on stable storage. When a guardian recovers from a crash, it increments its crash count, records the new value on stable storage, and adds the new crash count to its *map*. The guardian's *map* is included in (almost) every message sent to other guardians.

Associated with each local action at a guardian is a *dlist* (short for *dependency list*),

a list of all guardians which, if they crashed, would cause the action to become a crash orphan. The action is said to "depend on" these guardians. Essentially, an action becomes dependent on a guardian if it acquires data from the guardian, or modifies data there, and relies on the data continuing to be valid. Whenever a message is sent on behalf of an action, the action's *aid* and *dlist* are included in the message.

Maintaining the *dlists* is fairly simple. Initially, each action's *dlist* consists of its own guardian and any guardians that its parent depended on when the action was created. When one of the action's children commits, the child's *dlist* is merged (unioned) with the parent's. If a child aborts, its *dlist* is discarded. An action can also become dependent on a guardian by accessing an object for which one of its ancestors holds a lock; in this case, all guardians on which the ancestor depends at the time the lock is acquired are added to the descendant's *dlist*. The identity of these guardians is determined by querying the ancestor's guardian during the process of obtaining the lock.

When a guardian G receives a message containing a piggybacked *map*, it first destroys any local crash orphans it can detect. An action A is a crash orphan if there is a guardian H in A's *dlist* with a larger crash count value in the incoming *map* than in G's own *map*. (A became dependent on H when H had the lower crash count, and H has subsequently crashed). After all local orphans are destroyed, the incoming *map* is merged with G's own *map*. If the two *maps* contain different crash counts for the same guardian, the larger value is used in the merged *map*. Finally, G determines whether the sending action is an orphan, by examining the crash count values for guardians in the sender's *dlist*. If the sender depends on a guardian whose crash count in G's *map* is larger than in the sender's *map*, the sender must be a crash orphan; in this case a refusal message is sent to the sender's guardian. If the sender is not an orphan, the message is processed normally.

Each guardian saves its *map* and *done* on stable storage whenever it participates in two-phase commit. This is necessary to ensure that the guardian can be restored to a consistent state when it recovers from a crash.

3.2 Central Server Optimization

There is a serious problems with the orphan detection scheme as described so far: *map* and *done* become very large. *Map* contains a crash count for every guardian that has ever existed, and *done* contains the *aid* of every action that has ever aborted. Information in *map* and *done* is never deleted, so these data structures can grow without bound. In the present Argus implementation, *map* and *done* entries are deleted after an arbitrary period of time (one to four hours). The system does not guarantee that information will be retained as long as it is needed, but after such a long period of time, the probability of active orphans is considered acceptably small. Even so, because *map* and *done* are included in (almost) every message between guardians, an unreasonable amount of overhead is produced.

One way to reduce the overhead associated with the Argus orphan detection algorithm is to store orphan detection information in a central server. This optimization is described in [6]. In this scheme, a central map server is used to hold a copy of the *map*. Each guardian has its own copy of the *map*, and a timestamp, supplied by the server, indicating how recent that copy is. Only these timestamps, not the entire *map*, are exchanged in messages between guardians. This reduces the cost of inter-guardian communication. Guardians can obtain a new *map* fron the server when necessary. If guardian crashes are relatively rare, the *map* should not change very frequently, so guardians should not have to talk to the server too often.

When a guardian recovers from a crash, it informs the central server of its new crash count. The server records the new value in its copy of the *map*, saves the result to stable storage, increments the timestamp, and returns the new *map* and timestamp.

Each time a guardian receives a message, it compares the incoming *map* timestamp with its own. If they are the same, the sending and receiving guardians have the same *map*, and the message can be processed. If the receiving guardian's timestamp is newer, the sending action may be an orphan; if so, then the message should be refused. (The method

of determining whether the sender is an orphan is given below). If the sender's timestamp is more recent, the receiving guardian requests a new *map* and timestamp from the server. It uses the new *map* to detect and destroy local orphans and to determine whether the sending action is an orphan. If the sender is an orphan, the message is refused; otherwise it is processed normally.

In order to make it possible to determine whether a message was sent on behalf of an orphan, a small number of *map* entries are still included in messages. Each message includes the *map* entries for guardians in the sending action's *dlist*; this abbreviated *map* is called the *dmap* (short for *dlist map*). After getting a new *map* from the server if necessary, the receiving guardian compares the *dmap* entries with its own *map* entries for the same guardians. If any guardian has a lower crash count in the incoming *dmap* than in the recipient's *map*, the sending action is a crash orphan, and the message is refused by the recipient.

When a guardian is destroyed, the server is informed of this fact. The guardian's crash count is then set to a value representing infinity, larger than any integer. Eventually, the entry for this guardian is removed from the *map*; this prevents the *map* from growing without bound. If a guardian destroys itself (by executing a *terminate* statement), it notifies the server before it disappears. If the guardian is destroyed by the guardian manager, then the manager notifies the server.

In principle, a central server could be used to hold *done*, as well as *map*, thereby eliminating the need to send *done* in messages. However, because *done* changes much more frequently than *map* (i.e. every time an action aborts), guardians would need to communicate with the server too often.

Although it is not practical to stop sending *done* in messages, a central server based scheme can be used to reduce its size. When the server is used for this purpose, each guardian maintains a stable counter called a *generation count*, which it periodicallly increments. Whenever a (non-nested) topaction is created, it is tagged with its guardian's

generation count. Any descendants of this action are tagged with the same generation number. When an action aborts, its entry in *done* also includes this generation number. Periodically, each guardian checks to see which of its generations still have active local actions. If all actions from the guardian's first $N-1$ generations have completed (committed or aborted), then the server is notified of that fact. The guardian informs the server that the earliest generation that can have active actions is generation N.

A central generation server maintains a list of all guardians and their generation counts. This list is called the *generation map*, or *gmap*. When a guardian notifies the server that the Nth generation is the earliest that can have active actions, the server sets the guardian's *gmap* value to N. It then returns the new version of *gmap*, and a timestamp indicating how recent that version is. Each guardian maintains a copy of the *gmap*, along with its associated timestamp. The timestamp of a guardian's *gmap* is called the guardian's *generation timestamp*. The generation timestamp is included in (almost) all messages between guardians.

Guardians can use information about generations to eliminate *done* entries that are no longer needed. If a guardian's *gmap* value is N, then all top-level actions from that guardian's first $N-1$ generations have completed. Any active actions descended from those topactions must be orphans, and can be destroyed. Let us define the *origin* of an action as the guardian at which the action's non-nested topaction ancestor was created. If an action in *done* has a generation number less than the *gmap* value for its origin, that action's entry can safely be removed from *done*. Any of the action's descendants must belong to the same generation, and will be detected as orphans using the information in *gmap*.

When a guardian receives a message, it compares the incoming generation timestamp with its own. If the two timestamps are the same, the sending and receiving guardians have the same *gmap*, and the message can be processed normally. If the sender's timestamp is newer, the receiving guardian requests a new *gmap* from the server, and uses it to detect

local orphans, which are then destroyed. If the recipient's timestamp is newer, a new *gmap* is not requested. After comparing timestamps, and obtaining a new *gmap* if necessary, the receiving guardian determines whether the message was sent on behalf of an orphan. The sending action is an orphan if its generation number is less than the *gmap* value for its origin; in this case the message must be refused.

Because the central server is used to hold the *map* and *gmap*, it is not necessary for each guardian to keep a copy of them on stable storage. Instead, each guardian keeps a stable copy of its timestamps. Timestamps are saved to stable storage when the guardian participates in two-phase commit. During recovery from a crash, the guardian queries the server to obtain versions of *map* and *gmap* at least as recent as it had before; as usual, the server returns the new *maps* with timestamps indicating just how recent they are.

3.3 Replication of the Server

If guardians must rely on a central service in order to perform orphan detection, it is important to prevent the server from becoming a bottleneck. If the server was implemented as a single guardian, the entire system would stop working if that guardian ever crashed. To avoid this problem, the server is replicated at several guardians (e.g. three to seven).

To maintain the consistency of data stored at different replicas, we use a scheme based on the method described in [6]. In our implementation, it is possible to store data by communicating with any replica. In order to retrieve information, it may be necessary to communicate with one or more replicas. A "front end" for the map service or generation service, running at the client's guardian, queries as many replicas as necessary to obtain data recent enough to satisfy the client's needs. Each replica queried returns its version of the data, as well as a timestamp indicating how recent that version is. When the front end for the map service obtains sufficiently recent information, this information is then returned to the client.

Replicas periodically send each other "gossip" messages, to communicate changes in the

database. This reduces the number of queries the front end has to make. If gossip messages are sent often enough, the front end will generally be able to get the information it needs from the first replica it tries.

When this replication method is used, *map* versions can only be partially ordered: when each of two replicas possesses information unknown to the other, neither of their *maps* can be considered more recent. Some method is needed to allow replicas to generate appropriate timestamps, reflecting this partial ordering. Each replica must be able to generate its own timestamps independently, and newer stamps must always be associated with newer information.

Multi-part timestamps [6] provide a solution to this problem. In this implementation, a timestamp is a sequence of integers, containing one element for each replica of the server. Each component of the stamp corresponds to a specific replica. One timestamp is considered greater than or equal to another if every component of the first stamp is greater than or equal to the corresponding component of the second stamp. The stamps are equal if all corresponding components are equal. In all other cases, the timestamps are not comparable. A replica may create new timestamps only by incrementing its own component of an existing stamp, or by merging two existing stamps, as explained below.

Each replica is initially created with a timestamp in which every component is zero. When a replica receives information from a client, it increments its own component of the timestamp. Upon acquiring information from another replica, it merges the other's timestamp with its own, by taking the larger value of each component, to obtain a stamp at least as recent as either of them. Thus, larger timestamps are always associated with newer information.

Each time a client guardian receives a message, it merges its own timestamp with the stamp contained in the message; the client then asks the service for information at least as recent as the merged timestamp. The guardian must obtain new information from the service unless the incoming timestamp is less than or equal to the guardian's own stamp.

Client guardians are not allowed to manufacture arbitrary timestamps. A client may only create a zero timestamp, use a timestamp created by the service, or merge two properly created stamps. This ensures that each timestamp possessed by a client corresponds to information already in the service.

The above implementation of the server can be used in by our orphan detection scheme, because guardians do not need the most recent version of *map* and *gmap*. It is sufficient for a guardian to obtain any *map* and *gmap* whose timestamps are at least as recent as the corresponding timestamps included in the incoming message.

Chapter 4

Map Service Overview

The organization of the map and generation map services is similar to that described in [6]. Client guardians communicate with the map server through an abstract data type called *map_service*. The *map_service* provides all necessary operations for modifying and retrieving map entries, while hiding the replication of the server from the client. An essentially identical data type, called *gen_service* provides access to the generation map server. Several calls to *map_service* and/or *gen_service* operations may run concurrently, without interfering with each other. The server replicas themselves are implemented as guardians. Each replica provides operations to add and delete map entries, to return the replica's current map version, and to process gossip from other replicas. Replica operations are called only by the *map_service* data type, or by other replicas, never directly by the client.

4.1 Specification of the Map Service

The specifications for *map_service* are given in Figure 4.1. Because this type of service could be used in many applications, not just for orphan detection, the specifications are given in general terms. The map service is modeled as a set of states associating unique ids (uids) with integers. Each state corresponds to a version of the map, associating guardian ids with their crash counts. Every state has an associated timestamp; states with larger timestamps have newer information. In the specification of the service, the notation $s(u)$ refers to the value associated with uid u in state s; $s.ts$ stands for the timestamp of state s.

map_service = data type is insert, delete, refresh

Overview:

The map_service associates uids with integers. It consists of a set Z of states, each marked with a unique timestamp. Each state maps uids to integers. Initially, there is a single state in which all uids are mapped to $-\infty$. Insert and delete may cause new states to be added to Z. States with larger timestamps associate larger values with the uids: if t_1 and t_2 are timestamps and s_1 and s_2 are their associated states, then $t_1 \leq t_2 \Rightarrow \forall u{:}uid, s_1(u) \leq s_2(u)$; all operations must preserve this invariant.

The following procedure is not actually implemented; it is a "pseudo-operation" used to define the behavior of insert and delete:

enter = proc (u: uid, x: int) returns (timestamp)
  requires: $\forall s \in Z, s(u) < \infty$
  modifies: Z
  effects: Adds zero or more new states s' to Z, each with $s'(u) = x$. Each new state is derived from some existing state s in which $s(u) < x$ and $s(v) = s'(v)$, for all $v \neq u$. Returns the timestamp t^* of some state s^* with $s^*(u) \geq x$. This state is either one of the new states added to Z or an existing state. If there is not already a state in Z with $s(u) \geq x$, then at least one new state is added. For each state added to Z, the associated timestamp is created in a way that satisfies the invariant. This operation is atomic.

Operations:

refresh = proc (t: timestamp) returns (state, timestamp)
  effects: Returns $(s, s.ts)$, where $s \in Z$, $s.ts \geq t$. This operation is atomic.

insert = proc (u: uid, x: int, t: timestamp) returns (state, timestamp)
  requires: $\forall s \in Z, s(u) < \infty$
  modifies: Z
  effects: equivalent to refresh(merge_stamps(enter(u,x),t)). The insert operation is composed of atomic steps but is not atomic itself.

delete = proc (u: uid, t: timestamp) returns (state, timestamp)
  modifies: Z
  effects: equivalent to insert(u,∞,t)

end map_service

Figure 4.1: Specifications for the Map Service

In order to express the *map_service* specifications more clearly, we define a "pseudo-operation" called *enter*. This operation is not actually implemented; it is used only to define the behavior of other operations. The *enter* operation increases the value associated with a uid u to a given value x. If the map service does not contain a state s with $s(u) \geq x$, then this operation creates a new state s' with $s'(u) = x$ and adds it to the service. The new state is derived from an existing state with a smaller value for u; in the new state, all uids except for u have the same values as before. Many such states may be created. New states with $s'(u) = x$ may be created even if the map service already contains a state in which $s(u) \geq x$. This operation returns the timestamp t^* of a state s^* reflecting the change to the map. The state s^* is either a new state, created by this operation, in which $s^*(u) = x$, or an existing state with $s^*(u) \geq x$. The *enter* operation is atomic.

The *map_service* data type provides three operations that can be used by clients. These are called *refresh*, *insert* and *delete*. The *refresh* operation is used to obtain a new version of the map. It takes a timestamp as an argument; this timestamp indicates the earliest version of the map that the client can use. *Refresh* does not add states to the service; it simply returns an existing state and its associated timestamp. The *refresh* operation is atomic.

The map service also provides an *insert* operation, which clients use to increase the value associated with a uid, and to obtain a state and timestamp reflecting the change. The *insert* operation is equivalent to an *enter* followed by a *refresh*. *Insert* takes three arguments: a uid u and value x to be supplied to *enter*, and a timestamp t indicating the earliest state the client is willing to accept. The timestamp returned by *enter* is merged with the supplied stamp t, creating a stamp at least as large as each of them. The merged stamp then becomes the argument to *refresh*. Although *insert* is composed of two atomic operations it is not atomic itself. If two *insert* operations are run concurrently, each of them may return a state reflecting both changes to the map.

A client can delete a uid from the map by using the *delete* operation. This procedure maps the uid to a value representing infinity, larger than any integer. The *delete* operation

is equivalent to an *insert* with a value of infinity. Like *insert*, this operation is composed of two atomic steps, but is not atomic itself.

The specifications for the map service operations are non-deterministic. Non-determinism is introduced because operations may be performed at any replica of the server; different replicas may have different versions of the map. Any version with a sufficiently recent timestamp is acceptable to the client, so there is no need to contact every replica. Therefore, there is no guarantee as to precisely which state will be returned.

4.2 Replica Specifications

Figure 4.2 gives the specifications for the server replicas. In the figure, the notation $t[i]$ refers to the ith component of timestamp t. As in the case of the map service, $s(u)$ is the value of uid u in state s.

The *server replica guardian type* provides a creator called *new*, which constructs a new replica with an empty state and zero timestamp, and operations to insert and delete uids and to return the replica's state and timestamp. Each server replica has an *insert* operation, which adds a uid u to the replica state with a given value x, and returns the resulting state and timestamp. If u already has a value greater than or equal to x, then the state is not changed and the current state and stamp are returned. Replicas also have a *delete* operation, which sets the value of a given uid to ∞, and returns the resulting state and stamp. If the uid already has a value of ∞, then no changes are made and the current state and stamp are returned. A *refresh* operation is also provided by each replica; this operation returns the replica's current state and timestamp. In addition to the above operations, each replica has a *receive_gossip* routine, which is used to process gossip from other replicas. This routine takes a state and timestamp from a gossip message and merges them with the replica's current state and stamp.

The specifications for the replica operations are somewhat different from those of the corresponding routines of the *map_service* data type. The replicas' *insert*, *delete* and *refresh*

server_replica = **guardian is new**
with operations insert, delete, refresh, receive_gossip

Overview:

A mutable mapping from uids to integers. A replica r is $\langle i, s, t \rangle$, where s is a state mapping uids to integers, t is a timestamp associated with s, and i is an integer identifying this replica of the server. Replica states with larger timestamps associate larger values with the uids: if t_1 and t_2 are timestamps and s_1 and s_2 are their associated states, then $t_1 \leq t_2 \Rightarrow \forall u : uid, s_1(u) \leq s_2(u)$; all operations must preserve this invariant. All server replica operations are atomic.

Operations:

new = **creator** (i: int) **returns** (server_replica)
effects: returns $\langle i, s, t \rangle$, where t is a zero timestamp and $s(u) = -\infty$ for all u.

insert = **proc** (u: uid, x: int) **returns** (state, timestamp)
requires: $s(u) < \infty$
modifies: r
effects: sets r equal to $\langle i, s', t' \rangle$, where $s'(u) = \max(s(u), x)$; $s'(v) = s(v), \forall v \neq u$; if $s(u) \geq x$, then $t' = t$; otherwise $t'[i] > t[i]$ and $t'[j] = t[j], \forall j \neq i$. Returns ($s'$, t').

delete = **proc** (u: uid) **returns** (state, timestamp)
modifies: r
effects: sets r equal to $\langle i, s', t' \rangle$, where $s'(u) = \infty$; $s'(v) = s(v), \forall v \neq u$; if $s(u) = \infty$, then $t' = t$; otherwise $t'[i] > t[i]$ and $t'[j] = t[j], \forall j \neq i$. Returns ($s'$, t').

refresh = **proc** () **returns** (state, timestamp)
effects: returns (s, t)

receive_gossip = **proc** (s_g: state, t_g: timestamp)
effects: sets r equal to $\langle i, s', t' \rangle$, where $\forall u, s'(u) = \max(s(u), s_g(u))$ and $t' =$ merge(t, t_g).

end server_replica

Figure 4.2: Server Replica Specifications

operations do not take timestamps as arguments; they simply modify the current state, if necessary, and return the new state and timestamp. All replica operations are deterministic; they operate on what ever state the replica currently has.

Unlike ordinary guardians, server replicas have "operations" rather than handlers. These operations are simply procedures that run at the replica. Rather than making handler calls, the *map_service* data type sends low-level system messages to the replicas, instructing them to invoke these operations and send back the results. The reasons for avoiding handler calls for communication with replicas are given in Chapter 5.

4.3 Implementation

In this section, we present a simplified description of the implementation of the *map_service* data type and of the server replicas. A more detailed discussion of this implementation is given in the next chapter.

4.3.1 Map Service Operations

When the *map_service* data type's *insert* operation is called with uid u, value x, and timestamp t, the *map_service* sends an *insert(u, x)* message to any replica of the server. This instructs the replica to invoke its own *insert* operation, with arguments of u and x; the resulting state and timestamp are then returned to the *map_service.* If the returned timestamp t' is greater than or equal to the requested stamp t, then the *map_service* returns the map and stamp to the client. Otherwise, the *map_service* sends *refresh* messages to other replicas, one at a time, to obtain their states and stamps. These states and timestamps are merged with those already received. Eventually a sufficiently recent map and stamp are obtained, and these are returned to the client.

When a replica receives an *insert(u, x)* message, it invokes its *insert* operation as requested. When this operation is executed, the replica looks up u in its version of the map

to determine whether its value is less than x. If so, the replica sets the value of u to x, and increases its timestamp, by incrementing the stamp component that it is allowed to change. If the value of u is greater than or equal to x, then the map and timestamp are not changed. In either case, the replica's new map and timestamp are returned to the *map_service*.

The implementation of the *delete* operation is similar to that of *insert*. The *map_service* sends a *delete(u)* message to a replica, instructing the replica to run its own *delete* operation with argument u. The resulting state and timestamp are sent back to the *map_service*. If the new timestamp returned by the replica is at least as large as the stamp supplied by the client, then the *map_service* returns the new state and stamp. If the new stamp is not large enough, then the *map_service* sends *refresh* messages to other replicas, until sufficienly recent information has been obtained.

When a replica executes its *delete* operation with an argument u, it determines whether uid u has already been deleted (i.e. whether its value is already equal to ∞). If not, the map is changed, and the timestamp is incremented. In either case, the map and timestamp are returned to the *map_service*.

To perform the *refresh* operation, the *map_service* sends a *refresh* message to any replica. This causes the replica to invoke its own *refresh* operation. The replica sends its map and timestamp to the *map_service*, which returns them to the client if the stamp is large enough. Otherwise, the *map_service* queries additional replicas until sufficiently recent information is obtained.

Periodically, each replica sends a gossip message to each of the other replicas. This message contains the replica's current map and timestamp. When a replica receives a gossip message, invokes the *receive_gossip* operation. This routine first examines the message's timestamp. If the message timestamp is less than or equal to the replica's stamp, then the message contains no new information and is ignored. Otherwise, the replica merges the incoming map with its own. If a uid appears in either of the two maps, its value is included in the merged map; if a uid has different values in the two maps, the larger value is used.

The incoming timestamp is then merged with the replica's stamp, by taking the larger value of each component.

4.3.2 Removing Deleted Entries

As discussed in [6], it is necessary to be careful in dealing with entries for guardian ids that have been deleted from the database (i.e. map entries with a value of ∞). Clearly, we do not want to retain these entries forever. However, if these entries are simply removed from the map, the server cannot distinguish uids that have been deleted from uids that never were entered at all. If a server replica removes an entry from the map and later receives a message to insert a new entry for the same uid, the insert message should be ignored, but the replica has no way of knowing this.

Another problem involving deleted entries occurs if one replica of the server removes an entry from its map before the others learn that the entry is being deleted. One of the other replicas could then send a gossip message to the first, containing a finite value for the same uid. Again, the first replica would not be able to determine whether the uid should be inserted. Thus, a replica cannot delete an entry unless it can be sure that no more enter messages or gossip messages will be received that contain finite values for the same uid.

In our implementation of the server, we solve these problems using the method described in [6]. First, we require that no client can insert an entry for a uid into the map after that uid has been deleted. When the map service is used for orphan detection, each guardian inserts and deletes only its own id, so this constraint should never be violated. However, because of the possibility of network failures, messages may be delayed or delivered out of order. Therefore, it is still possible for a replica to receive an insert message after a delete message for the same uid.

To deal with late insert messages, a replica discards any message received more than a certain time after it was sent. Each message from the *map_service* to a replica contains the

time at which the message was sent, as measured by the sender's clock. Let τ_r be the time a message is received, τ_s the time it was sent, and δ the largest apparent delay after which we will still accept a message. Any message for which $\tau_r - \tau_s > \delta$ will be discarded. Now suppose that a replica receives a delete message that was sent at time τ_d, according to the sender's clock. An insert message for the same uid could have been sent by the same client as late as τ_d, but no later, and will be discarded if received after $\tau_d + \delta$. Thus, the entry for this uid should not be removed until $\tau_d + \delta$.

If we assume that guardians' clocks are loosely synchronized, with maximum skew ϵ, the actual time this scheme allows for delivery of a message will vary between $\delta - \epsilon$ and $\delta + \epsilon$. Therefore, δ should be chosen large enough to allow most messages to be delivered, but small enough to avoid excessive delays in garbage collection.

Our method of removing deleted entries also assumes that clocks are never set back. If a guardian's clock ever was set back, the *map_service* implementation at that guardian could send a server replica an *insert* message having a later time than a subsequent *delete* message for the same uid. If the guardian's clock must be set back, the time values included in its map service messages must not change, until the clock reaches its previous maximum value.

If we allow more than one client to perform insert operations for the same uid, as long the uid is not inserted after it is deleted, then deleted entries must be retained until $\tau_d + \delta + \epsilon$, not just until $\tau_d + \delta$. This allows for the possibility that the inserter's clock is slower than the clock of the client making the deletion.

To allow garbage collection of deleted entries, each replica maintains a table of map entries that have been deleted (i.e. set to an infinite value) but not actually removed. Each entry in this table contains the uid of the deleted entry, the time at which the delete message for the entry was sent, and the replica's timestamp after executing the delete operation. Each replica also maintains a table of the other replicas' timestamps, indicating the least recent timestamp each replica could possibly have. Periodically, the replica examines the

table of deleted entries, to determine which of them can be removed from the map and from the table itself.

An entry cannot be removed until all of the server replicas have learned that the entry has been deleted. This is determined by comparing the entry's timestamp with the timestamps of the replicas. The entry cannot be purged unless every replica's timestamp is greater than or equal to the stamp of the entry, ensuring that every replica knows of the change. In addition, an entry must be retained until the current time at the replica is later than the time of the entry plus the maximum message delay δ. Once both of these conditions are satisfied, the entry can be purged.

4.3.3 Optimizing Gossip Messages

It is not necessary for a replica to send the entire map in gossip messages. Instead, the replica can maintain a list of map entries that the other replicas may not have seen. Each time a change is made to the map, the new map entry can be added to this list, along with the replica's timestamp after making the change. An entry need not be sent to a replica if the entry's timestamp is less than or equal to the recipient's. When all replicas' timestamps are greater than or equal to the entry's stamp, the entry can be removed from the gossip list. The other replicas' timestamps can be estimated based on the timestamps they have sent in their own gossip messages; these estimates will be lower bounds, so information will not be removed from the gossip list too soon.

When gossip is handled in this manner, each gossip message includes the following information: the identity of the sender, the timestamp of the sender after all entries in the gossip set were added to the map, and the gossip entries themselves. Each entry consists of a guardian id, its new map value, and the timestamp value when the update was processed. In addition, a deleted entry (i.e. an entry with a value of infinity) contains the time at which the client sent the delete message. Replicas use the timestamp values to determine which entries to pay attention to; any entry with a stamp earlier than the replica's own stamp can

be ignored. Both time and stamp values are used to determine when deleted entries can be garbage collected.

Gossip messages must include a timestamp for each deleted entry. This enables the recipient to add the entry to its deleted entry table with an appropriate stamp. If the recipient simply used its current timestamp, the entries' stamps could become later and later, and the entries might never be garbage collected.

It is not strictly necessary for gossip messages to include the time values for each entry. To allow deleted entries to be garbage collected, each replica could construct a deleted entry containing the time at which it received the entry plus ϵ, the maximum clock skew. These time values will be later than the original delete time, so deletion cannot occur too soon. The time values cannot get later forever; at some point every replica will have a timestamp later than the entry's stamp, and it will no longer be sent in gossip messages. Thus, the entry will eventually be garbage collected. However, garbage collection of the entries could be delayed by an amount equal to the maximum skew.

Gossip messages need not include timestamps for insert entries. Because the value associated with a guardian id can never decrease, a replica could simply ignore a gossip entry unless the value it specified was higher than the replica's own value for the same id. If a replica has already garbage collected a deleted entry for a uid u, then the sender must have learned about the deletion, and subsequently gossiped to the recipient. In this case, the recipient's timestamp will now be later than the sender's stamp was at the time the sender learned of the deletion. If a message subsequently received from the sender has a later stamp, that message must contain an infinite value for u. Any message with a finite value for this uid must have an earlier stamp, so the whole message will be ignored by the recipient. If this optimization is used, the recipient can still garbage collect the gossip list properly. When a gossip message is processed, and new insert entries are added to the recipient's gossip list, the entries can be tagged with the timestamp of the incoming gossip message, allowing garbage collection when all replicas' timestamps are larger than

this value.

The above discussion assumes that finite and infinite values for a given uid are never sent in the same gossip message. To ensure this, each replica should remove finite valued entries for a uid from the gossip list when an infinite value for that uid is inserted, and should not add a finite value for a gid whose value is known to be infinite.

4.3.4 Reconfiguration

The map server design should allow for the possibility of reconfiguration of the service. From time to time, both the number and location of server replicas may changes. In the present implementation, installing a new configuration requires shutting down the map service temporarily, while the new replicas are constructed. Reconfiguration involves the following steps:

1. Each replica must be told to stop processing client requests.
2. The states of all replicas must be obtained and merged.
3. New replicas must be constructed, each containing the new state.

The new replicas may then begin processing clients' requests.

It is necessary to have some way for replicas and clients to interpret timestamps from different configurations. Our implementation deals with this problem as suggested in [2]. Each timestamp contains a configuration number identifying the map service configuration in which it was generated. Later configurations are assigned higher configuration numbers. Any timestamp from a later configuration is considered greater than any timestamp from earlier configurations. The merge of stamps from different configurations is equal to the later stamp. Because stamps from later configurations are assumed to represent newer information, it is necessary for a replica of the new configuration to have all information from the previous one.

To enable clients to locate the map server replicas, information concerning the new configuration is inserted into the Argus catalog. The catalog is a service that all guardians can use to locate resources in the Argus system.

It is undesirable for the map service to shut down during reconfiguration. If the server is reconfigured rarely, and if reconfiguration can be accomplished quickly, the temporary disruption of service might be acceptable. However, if some replicas cannot be contacted quickly, because of node crashes or network failures, reconfiguration could take too long.

A more sophisticated method of reconfiguration is discussed in [2]. This method allows replicas to be removed without disrupting service. When more general reconfigurations occur, some lookups can be processed; other lookups, and all updates, are blocked until reconfiguration is completed. Although this method could be adapted for our application, implementation of a more involved reconfiguration scheme is beyond the scope of this thesis.

Chapter 5

Implementation of the Map Service and Replicas

The map service has been designed to reduce the effects of two important sources of overhead: communication and stable storage. Sending and receiving messages and writing data to stable storage can take orders of magnitude more time than other operations, and have the potential to dominate other costs. The server design attempts to minimize both the information written to stable storage and the number and size of messages.

Another goal in the design of the server replicas is to allow as much concurrency as possible. Certainly, several clients should be able to read data concurrently from a replica. Ideally, it should even be possible for read operations to proceed at the same time as an update at the same replica. In this case the state and timestamp returned by the read operations would not reflect any of the changes made by the update.

5.1 Replica Implementation

Each replica of the server is implemented as an Argus guardian. This makes it possible to take advantage of the Argus stable storage and recovery mechanisms. Even though replicas are implemented as guardians, communication among replicas, or between replicas and the *map_service*, does not take place through handler calls. Instead, low-level system messages are used. This is done to eliminate the overhead associated with Argus handler calls (e.g. piggybacking of *aids* and orphan information; two phase commit for the call's topaction). In addition, if normal handler calls were used for this purpose, orphan detection would have

to be performed for them. The orphan detection routines might find it necessary to call the server again, resulting in an infinite loop.

The use of handler calls for communication with replicas can be avoided in our application, because the required atomicity of operations can be achieved without them. Operations within a replica either run to completion or have no effect. Replica operations are never aborted by the *map_service* nor are *map_service* operations ever aborted. In addition, these operations are never combined into larger atomic units.

Replica operations never have to be aborted by the *map_service*, because repeating any operation twice at a replica has the same effect as performing it just once. Performing an operation at several replicas instead of one also gives correct behavior. Therefore, if the *map_service* does not receive a reply for a request within a reasonable time, a second call can be made to the same replica, or a different one, without aborting the first call.

There is no need to use the Argus transaction mechanism to serialize replica operations. Operations performed at a single replica are serialized by the replica itself, as explained later in this chapter. There are no serial constraints on operations at different replicas, except that a gossip messaage reflecting an update must not be sent until the sender has recorded the update on stable storage. The sender of the gossip message can easily enforce this constraint; it is considered a "multiple replica" constraint only because gossip processing is considered an operation of the recipient.

Similarly, *map_service* operations never need to be aborted. Clients never deliberately abort *map_service* operations. If a client crashes before an update operation is completed, there is no need to undo the operation; the information inserted into the service remains valid. Note that some of our *map_service* operations are not serializable. If two guardians perform *insert* or *delete* operations at the same time, the state returned by the *map_service* for each of them could reflect the update made by the other. This can happen if the timestamp supplied by the client is large enough to require queries to other replicas to get a recent enough state, or if a replica's reply is lost and the update is retried, or if a

background query is made before the update completes. There is nothing wrong with this behavior, for our purposes.

In some applications other than orphan detection, it may be necessary for all *map_service* operations to be atomic and combinable into larger atomic units. An implementation of the server that allows operations to be combined into compound atomic actions, by making use of handler calls for server operations, can be found in [2].

5.1.1 Components of the Replica State

The state of a map server replica consists of the following elements:

map a volatile copy of the map data, available for lookup operations; it does not reflect any updates that are still in progress

current_timestamp the timestamp of the volatile map

stable map a copy of map, kept on stable storage; does not reflect the most recent map updates; periodically, the volatile map is flushed to stable storage, to update this stable copy.

changes a stable log of changes to the map; it consists of two components:

- **new_entries** a list of map updates not yet added to the stable version of the complete map. Together, the stable map and new_entries define the map version to be restored upon recovery from a crash.
- **timestamp** the timestamp of the state obtained by merging new_entries into the stable map; i.e. the timestamp to be used after crash recovery.

deleted_entries a set of guardian ids for which the delete operation has been performed, but which have not yet been garbage collected from the map. Each entry in this set is of the form (name, time, stamp), where name is the uid this entry is for, time is

the time when a client made the deletion request, and stamp is the replica timestamp when the delete operation was performed. Time and stamp are used to determine when deleted entries may safely be removed from the map.

replica_timestamps estimates of the timestamps of the other replicas; these estimates are lower bounds, based on the messages received from each replica.

gossip_list a list of map entries to be gossiped to other replicas. Each gossip entry is of the form ⟨name, value, time, stamp⟩, where ⟨name, value⟩ is a map entry, and time and stamp indicate when that entry was added to the map.

read_synch dummy object used for synchronization

write_synch another dummy object for synchronization

status a stable object describing the current server configuration, the location of the other servers, and indicating whether reconfiguration is in progress.

One goal in the design of the server was to minimize the use of stable storage, to avoid unnecessary overhead. For this reason, several elements of the replica state are volatile. These include the gossip list, the deleted entry set, and the estimates of other replicas' timestamps. The above items can be recovered to a safe, although non-optimal, state when the replica recovers from a crash.

When the list of estimated replica timestamps is reconstructed after a crash, each replica can be assigned a zero timestamp. This may cause us to send each replica some gossip entries it has already received, and may delay garbage collection. However, as soon as we receive gossip from a replica, we can update our estimate of its timestamp appropriately, so garbage collection should not be delayed for very long. An alternative solution is to ask the other replicas to send us their timestamps immediately; if a replica does not respond to our request, we can assign it a zero timestamp temporarily.

The deleted entry table can be reconstructed from the stable version of the map. An entry is created for each guardian id mapped to a value of infinity. Our current time and timestamp are used to construct each deleted entry. This ensures that entries are not garbage collected too soon. Because there should normally be only a few deleted entries, any delay in garbage collecting them should not be a problem.

Because the gossip list is not saved on stable storage, we cannot tell which map entries are already known to other replicas. A new gossip list is constructed containing all entries in the database, with time and stamp values equal to our current time and stamp. As a result, our gossip messages include the entire map for some time after recovery. The list is eventually garbage collected in the usual manner.

The decision not to make the gossip list stable was made because it seemed better to reduce performance somewhat after a replica crash than to write more to stable storage every time an update was made. However, for a sufficiently large database, gossiping all the data after a crash would not be acceptable. In this case, the list of gossip entries could be made stable. Only the list of guardian ids in the gossip list would have to be saved, however. Appropriate values for each id could be obtained from the stable map; safe values for the time and stamp of each entry could be calculated as described above.

5.1.2 Controlling Concurrency

Another goal in the design of the server replicas was to achieve as much concurrency as possible. In order to prevent the server from becoming a bottleneck, we want to prevent different operations at a replica from delaying each other. It seems particularly undesirable for operations that only read data from the server to be delayed by update operations; updates take longer to complete than reads do, because of the relatively long time needed to write to stable storage. It would be convenient to allow lookup operations to return immediately, with a version of the server state that does not reflect the updates in progress, thereby avoiding the delay.

In our present Argus implementation this is not possible, because writing to stable storage causes the entire guardian, not just the current process, to wait. The reason for this is that each Argus guardian runs as a single process in the underlying UNIX[1] system; concurrency at a guardian is achieved by creating multiple threads of control within this process, each corresponding to a logical process of the Argus system. In our present implementation, writing to stable storage blocks the underlying UNIX process, and therefore the whole guardian, until the write is completed. Because the server design is intended to be general, not restricted to the present Argus implementation, and because our stable storage implementation may be changed in the future, this problem was ignored. All further discussion in this paper will be based on the assumption that writing to stable storage will not block the guardian.

In order to allow concurrent reads and writes, two copies of the current state and timestamp are maintained. One copy is kept on stable storage; the other is volatile. Each *read operation returns* a copy of the volatile state and timestamp. Updates change the stable state and timestamp, wait for the change to be written to stable storage, and finally update the volatile state. The server also maintains a list called *changes*, listing the last few changes to the map. When the map is modified, only this list is written to stable storage, not the whole database. Periodically, the stable copy of the whole map is updated to reflect these changes, and the changes list is reset to be empty.

To synchronize access to the server state, two objects are used: an atomic object (named *read_synch*) and a mutex (named *write_synch*). These objects are used solely for synchronization; their values have no significance. *Read_synch* is used to control access to the volatile copy of the data; processes needing to examine or modify the volatile state must first acquire the appropriate lock on read_synch. *Write_synch* protects the stable data, and ensures that map information is written to stable storage only in a consistent state.

The following rules are used to avoid conflicts among operations:

[1]UNIX is a trademark of AT&T Bell Laboratories

1. Any operation modifying any part of the server state (or reading the *changes* list), must seize the *write_synch* mutex before before changing anything (or reading *changes*). The mutex must be held until all modifications have been made, and until *changes* has been accessed for the last time.

2. Any operation modifying any part of the replica state other than *changes* must acquire a write lock on *read_synch*, in addition to seizing the *write_synch* mutex. The lock on *read_synch* must be acquired before the first change is made and must be held until after the last change.

3. To access any part of the state, an operation must acquire a read lock on *read_synch* before the first access and hold it until after the last access, *except* that an operation does not need a read lock on *read_synch* while it has the *write_synch* mutex seized.

4. If an operation seizes the *write_synch* mutex, it must seize it before acquiring any other lock and hold it until after all other locks are released.

The first three of the above rules control concurrency in access to different components of the replica state; the fourth rule prevents deadlocks that might occur if two operations attempted to seize the same locks in a different order. The last clause of rule three allows lookups to proceed while the *changes* list is being written to stable storage. These rules can be suspended during recovery from a crash, when only the recovery process is active.

5.1.3 Implementing Replica Operations

Let us now examine the replica operations in detail. We present an "abstract implementation", ignoring some details of the Argus language.

The insert operation adds a map entry for a uid to the replica state. The state is unchanged if the uid already has an entry with a larger value. The insert routine ignores old requests, i.e. those for which the message delay is larger than the maximium allowed. This operation is implemented as follows:

```
insert = proc (name: uid, new_value: int, sender: client, send_time: time)

  if too_old(send_time) then return end  % ignore old insert messages

  % block other writers
  seize write_synch do

    % don't change the state if we already have a larger value
    % for name; if name has not been inserted yet, its value is -∞
    if map[name] < new_value then

      % update stable log of changes; flush log to stable storage

      % reset log if necessary
      if too_large(changes.new_entries) then
        flush_to_ss(map)
        changes.new_entries := {}
        end % if

      changes.new_entries := changes.new_entries ∪ {(name, new_value)}
      changes.stamp := increment_timestamp(current_timestamp,my_part)
      flush_to_ss(changes)

      % Now update the volatile data

      enter topaction

        % get a write lock to block readers from examining the state
        set_value(read_synch, dummy_value)

        % update the volatile state and timestamp
        map[name] := new_value
        current_timestamp := changes.stamp

        replica_timestamps[this_replica] := current_timestamp

        % add the change to the gossip list
        gossip_list := gossip_list ∪ {(name, new_value, send_time, current_timestamp)}

        end % topaction
      end % if

    reply_to_client(sender, map, current_timestamp)
```

```
    end % seize
  end insert
```

The delete operation is essentially equivalent to an insert operation performed with a value of ∞. It is implemented the same as *insert* except that the deletion must be recorded in the set of deleted entries. This is accomplished by executing the following immediately after updating the volatile state, and before changing the gossip list:

$$\text{deleted\_entries} := \text{deleted\_entries} \cup \{(\text{name}, \text{send\_time}, \text{current\_timestamp})\}$$

The following code implements the refresh operation:

```
refresh = proc (sender: client)

  enter topaction
    % get a read lock for the volatile state
    get_value(read_synch)

    reply_to_client(sender, map, current_timestamp)
    end % topaction

  end refresh
```

Each server replica contains a daemon responsible for sending gossip messages to other replicas. Every now and then, this process wakes up and sends each replica the list of changes to the state. A change is not included the gossip message if we know that the recipient has already learned about it, i.e. if our estimate of the recipient's timestamp is greater than or equal to the timestamp associated with the change.

```
gossip = proc ()
```

```
while true do
   sleep(gossip_interval) % wait until it's time to send next message
   seize write_synch do % block writers
     for next_replica:Server ∈ all_replicas do
         if next_replica ≠ this_replica then
           next_replica_gossip:SetOfGossipEntries := {}
           for e:Gossip_entry ∈ gossip_list do
               % don't send the entry if the replica already knows about it
               if (e.timestamp ≤ replica_timestamps[next_replica] then
                 next_replica_gossip := next_replica_gossip ∪ {e}
                 end % if
               end % for
           end % if
         send_to_replica(next_replica, next_replica_gossip, current_timestamp)
         end % for
     end % seize
   end % while

end gossip
```

When a replica receives a gossip message, it carries out the following steps. First, it revises its estimate of the sender's timestamp, by merging the old estimate with the timestamp of the gossip message. Next, if the message timestamp is less than the recipient's stamp, the message contains no new information and is discarded. Otherwise, each gossip entry is added to the stable copy of the map, and a new timestamp is calculated by merging the incoming timestamp with the recipient's own stamp. After these changes have been written stably, the volatile copy of the map and timestamp are updated, and the gossip entries are added to the gossip log. The implementation of this operation is given below.

```
receive_gossip = proc (msg: Gossip_message)

  % block other writers
  seize write_synch do

    % revise estimate of sender's timestamp
    replica_timestamps[msg.sender] :=
        merge_timestamps(replica_timestamps[msg.sender], msg.timestamp)
```

```
if msg.timestamp ≤ current_timestamp then

   % update stable log of changes; flush results to stable storage

   % reset log if necessary
   if too_large(changes.new_entries) then
     flush_to_ss(map)
     changes.new_entries := {}
     end % if

   % add the new map entries
   for e:Log_entry ∈ msg.new_entries do
       if e.timestamp ≤ current_timestamp and changes.new_entries[e.name] < e.value then
         changes.new_entries[e.name] := e.value
         end % if
       end % for

   % change the timestamp
   changes.stamp := merge(current_timestamp, msg.timestamp)

   flush_to_ss(changes)

   % now do the volatile state

   enter topaction
      % get write lock to block readers
      set_value(read_synch, dummy_value)
      % add each entry containing new information
      for e:Log_entry ∈ msg.new_entries do
          if e.timestamp ≤ current_timestamp and map[e.name] < e.value then
            map[e.name] := e.value

            % if entry is for a deleted uid, then add it to the list of deleted entries
            if e.value = ∞ then
              deleted_entries := deleted_entries ∪ {(e.name, e.time, e.timestamp)}
              end % if
            gossip_list := gossip_list ∪ {e}
            end % if
          end % for
      current_timestamp := changes.stamp
      replica_timestamps[this_replica] := current_timestamp
      end % topaction
```

```
    end % if
  end % seize
end  receive_gossip
```

In order to garbage collect information from the state, a replica periodically calls following routine:

```
cleanup = proc ()

  % block other writers
  seize write_synch do

    % first do gossip list; remove entries received by all replicas
    for e:GossipEntry ∈ gossip_list do
        if ∀ts ∈ replica_timestamps, e.timestamp ≤ ts then
          gossip_list := gossip_list – {e}
          end % if
        end % for

    % now remove deleted entries

    purged:Bool := false
    current_time:Time := % system primitive
    enter topaction

      % get write lock on the volatile state; we're about to change it
      set_value(read_synch, dummy_value)

      % remove an entry only if it's old enough and all other replicas know about it

      for e:DeletedEntry ∈ deleted_entries do
          if old(e.time) and ∀ts ∈ replica_timestamps, e.timestamp ≤ ts then
            remove_entry(changes.new_entries, e.name)
            remove_entry(map, e.name)
            deleted_entries := deleted_entries – {e}
            end % if
          end % for
      end % topaction

    % finally, flush modified map and changes list to stable storage
```

```
    flush_to_ss(map)
    flush_to_ss(changes)
    end % seize

  end cleanup
```

When a replica recovers from a crash, it must reconstruct the volatile state appropriately based on the stable data. The list of recent changes must be merged with the stable copy of the map, to produce a volatile map reflecting all data known to the server. The volatile timestamp is set equal to the stable timestamp. Some components of the replica state (e.g. gossip list, list of other replicas' timestamps, deleted entry list) are not saved stably; these are recovered to a safe, but not necessarily optimal state. After updating the state and creating background processes for gossip, garbage collection and communication with clients, the replica may begin performing normal operations. The recovery operations are given below.

```
recover % recovery process

  % make dummy objects for synchronization
  read_synch := ...
  write_synch := ...

  % Recover map and timestamp; volatile map should reflect all updates
  % in the changes list, so merge them in; take the opportunity to
  % reset the changes list and flush the whole map to stable storage.

  map := merge(map, changes.new_entries)
  current_timestamp := changes.timestamp
  flush_to_ss(map)
  changes.new_entries := {}
  flush_to_ss(changes)

  % we don't remember the other replicas timestamps;
  % it is safe to pretend they're zero
  for replica:Server ∈ all_replicas do
      replica_timestamps[replica] := make_zero_timestamp()
      end % for
```

```
replica_timestamps[this_replica] := current_timestamp

% recover deleted entries and gossip log; choose safe
% deletion times and timestamps for each entry
current_time := % system primitive
gossip_list := {}
deleted_entries := {}
for e:Map_entry ∈ map.value do
    gossip_list := gossip_list ∪ {(e.name, e.value, current_time, current_timestamp)}
    if e.value = ∞ then
      deleted_entries := deleted_entries ∪ {(e.name, current_time, current_timestamp)}
      end % if
    end % for

% fork processes for gossip, etc.

...

end % recover
```

It can be demonstrated that this implementation of server operations guarantees the consistency of the replica state, and of data returned to the client. If a lookup operation holds a read lock, writers will be excluded from modifying the volatile state, and a consistent view of the data will be obtained. The use of *write_synch* ensures that only one write can be in progress at a given time, so each writer will see a consistent state. To ensure the consistency of stable data, a single object (i.e. *changes*) is used to hold the new_entries list and timestamp; these are written to stable storage in a single operation. The volatile map is not updated until after the new map entry has been saved stably. This ensures that no client will see information that is still volatile and could later be lost. Periodically, the replica resets the changes list to be empty. Before this is done, all changes are first written to the stable copy of the map, to ensure that no changes are lost in the event of a crash.

Operations cannot deadlock with each other. Once a read operation has acquired the necessary read lock, it can always complete. Updates cannot deadlock with each other, because only one can be in progress at a time.

Update operations delay readers only for the time needed to modify the volatile map. While an update is writing information to stable storage, it does not hold a write lock on the volatile state, so readers are not delayed.

5.2 Implementation of the Front End

The implementation of the front end for the map service is fairly straightforward. As explained in Chapter 4, an abstract data type called *map_service* is maintained at each client; the client obtains map information by calling the operations of this type. The *map_service* data type maintains a queue of client requests and its own copy of the map and timestamp. The map and stamp are cached locally for greater efficiency; when sufficiently recent information is available locally, calls to the server replicas can be avoided. Periodically, the *map_service* queries replicas in background mode to obtain a new version of the map, so this information will be available the next time the client makes a request.

When the map_service's *refresh* operation is called, the timestamp supplied by the caller is compared to the map_service's stamp. If the timestamp of the caller's request is less than or equal to the cached stamp, then the local copy of the map and stamp are returned. Otherwise, the new request is added to the request queue. When the *insert* or *delete* operation is called, the request is added to the queue regardless of the value of the timestamp.

A separate process within the *map_service* handles the map requests, making the appropriate calls to the server replicas. When a response is received from a replica, this process merges the incoming state and timestamp with the local copies. Next, the local timestamp is compared with the timestamp of each client request. If any requests have a timestamp less than or equal to the new stamp of the local map, the processes making the requests are woken up, and allowed to return with the new information. If there remain unsatisfied requests after a response is received from a replica, or if the replica to which a request is addressed does not respond, another replica is tried.

In order for the above implementation to work correctly, it is necessary that the

map_service be able to merge information received from a replica with information cached locally. Specifically, the *map_service* relies on the following fact:

$$\forall s_1, s_2, s_3{:}state,\ s_3.ts = merge(s_1.ts, s_2.ts) \Rightarrow s_3 = merge(s_1, s_2)$$

Although the *map_service* specifications do not guarantee this property to client applications (to avoid constraining the implementation unnecessarily), the replica specifications must provide such a guarantee to the *map_service* data type itself. Our map server replicas preserve the above property, because of the way gossip messages are processed. When a gossip message is received by a replica, the recipient's new timestamp is set equal to the merge of the old and incoming stamps; the recipient does not increment its component of the stamp.

Chapter 6

Orphan Detection Using the Central Server

In this chapter, we will examine the implementation of our orphan detection method. First, guardian creation and destruction, crash recovery, and handler call processing will be considered as they relate to orphan detection. Next, we will discuss the implementation of modules used to detect crash orphans and abort orphans. Finally, we will explain how to destroy an orphan once it has been detected.

6.1 System Overview

Before discussing the implementation of our orphan detection algorithm in detail, it is useful to describe briefly the structure of the Argus system. Only those portions of the system related to our orphan detection implementation will be discused here. The structure of this part of the system is illustrated in Figure 6.1. In this diagram, a plain rectangle indicates a procedure, one with a double line at the top indicates an abstract data type, and a box labelled with the letter "G" in one corner indicates a guardian. Arrows indicate dependencies among modules. The figure reflects the changes to the system resulting from use of the central server for orphan detection; previously, the system structure was similar, except for the absence of our new modules.

At the highest level of the system that we will consider, there are two modules: *sys_init* (for *sys*tem *init*ialization) and *handlertype*. *Sys_init* runs when a guardian or program is first created, or when a guardian recovers from a crash. It calls many other modules to

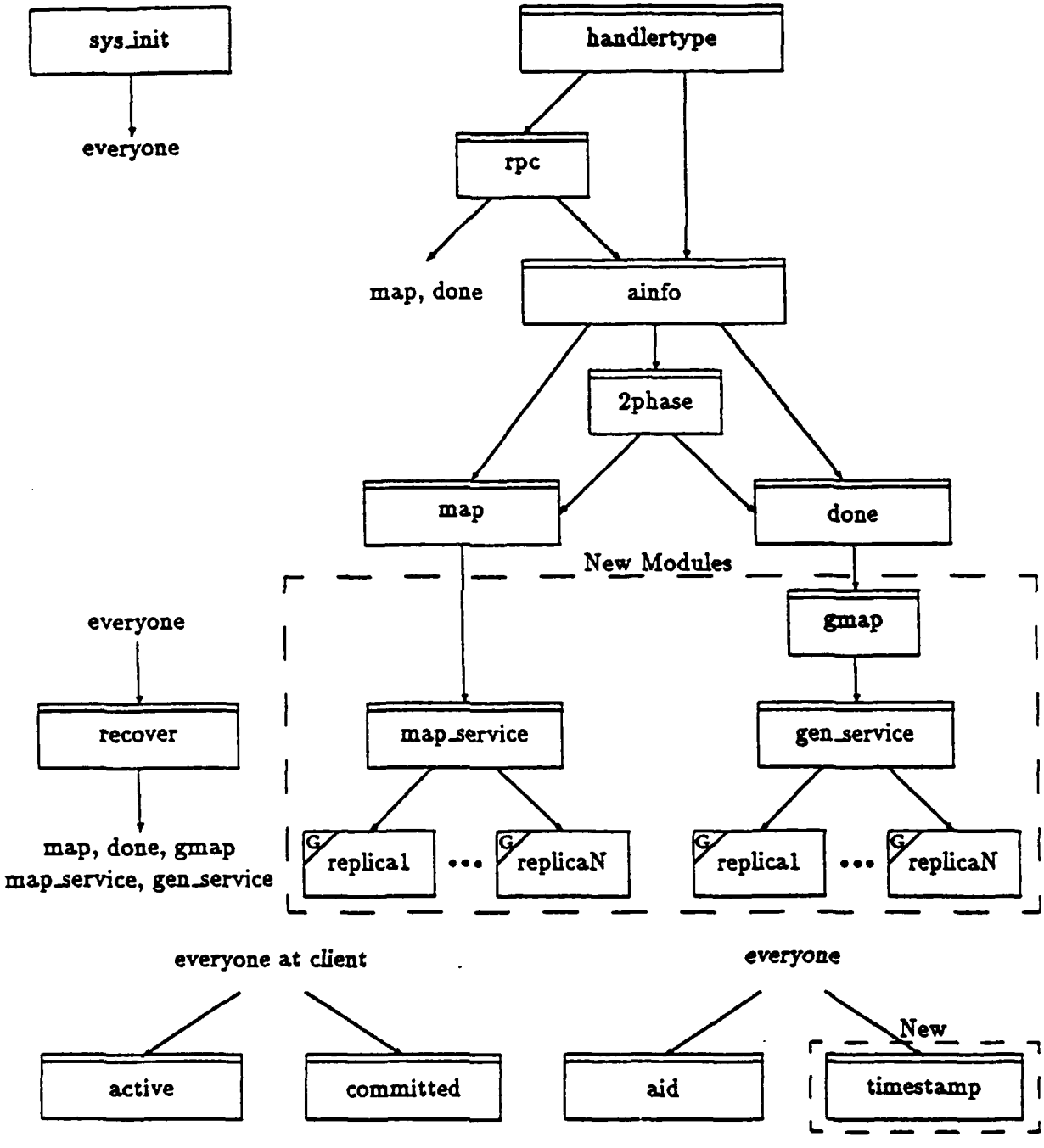


Figure 6.1: Argus Runtime System — Orphan Detection Modules

initialize data and to recover information from stable storage. *Handlertype* is an abstract data type used to implement handler calls. It performs the operations needed to send a handler call, run the call at the remote guardian, and process the reply. A subordinate module, *rpc*, waits for the reply to a handler call. Until a reply is received, *rpc* periodically contacts the recipient of the handler call to verify that it has not crashed, has not been destroyed, and that the call is still active. If the handler call is no longer active (e.g. because it was destroyed as an orphan), or if the guardian making the handler call is unable to communicate with the recipient, *rpc* reports the error to *handlertype*.

At the next level of the system is the *ainfo* (for action *info*rmation) module. *Ainfo* is used to create new actions, abort and commit them, and to add orphan information (as well as additional information used later for two phase commit) to handler call and reply messages. *Ainfo* is also responsible for reading and processing this information when a guardian receives a handler call or reply. The *2phase* module is called by *ainfo* when a topaction commits; it is responsible for carrying out the two-phase commit protocol [1].

The third layer in our system consists of those modules directly responsible for performing orphan detection. These are the *map*, *done*, and *gmap* data types. *Map* maintains a list of guardians and their crash counts, and is used to detect crash orphans. *Done* contains a list of aborted actions, and is used for detecting abort orphans. The *gmap* module implements the generation map, which holds the generation counts of other guardians. *Gmap* is used to determine when information can safely be removed from *done*; it also shares the responsiblity for detecting abort orphans. The *map*, *done*, and *gmap* modules correspond to the data structures of the same names described in Chapter 3.

Below the main orphan detection modules are the *map_service* and *gen_service* data types. *Map_service* is the front end for the central map service. It is used to inform the service of the guardian's crash count, and to obtain new maps from the service. Similarly, *gen_service* acts as a front end for the generation service; it is used to store the generation count, and to obtain a new *gmap*. The *map_service* and *gen_service* contact replicas of

the map server and generation server, respectively, to store or retrieve the appropriate information.

There are two other data structures maintained at each guardian that are relevant for orphan detection: *active* and *committed*. *Active* is a list of all local root actions that have not yet committed or aborted. (A local root is a non-nested topaction or a handler action, i.e. the root of an action tree at this guardian.) *Committed* lists all local roots that have committed, but not yet committed through the top. Many modules need to refer to the list of active actions. *Committed* is used to hold information about local committed actions until two phase commit can be performed for them; it is also used to identify and destroy "unwanted committed subactions", as explained later in this chapter.

Finally, at the lowest level, are the *aid* and *timestamp* data types. *Aid* implements action identifiers; *timestamp* implements the stamps associated with the map and generation map.

Outside of this layered structure is the *recover* module, used for stable storage and crash recovery. Most of the other modules make use of *recover* to write information to stable storage. During recovery, procedures within the *recover* module read the saved data, and call operations of *map*, *done*, *gmap*, and other data types, to restore those objects to appropriate states.

6.2 Guardian Creation, Destruction and Crash Recovery

When a guardian is created, a start up routine in the *map* module inserts the guardian's *gid* into the map service, with a crash count of zero, and obtains a map and timestamp from the service. This is accomplished by calling the map service's *insert* operation, with the guardian's *gid*, a zero crash count, and zero timestamp as arguments. The map and timestamp returned by the service are then used as the guardian's map and stamp. Similarly, a *gmap* routine inserts the *gid* into the generation service, with a generation count of zero, and obtains a copy of the generation map and generation timestamp. The *done* list is initially empty.

When a guardian recovers from a crash, the *recover* module uses the information saved on stable storage to reconstruct a consistent guardian state. The crash count is read and incremented, and the new crash count is saved. The previously saved generation count is also obtained. The map and generation timestamps are recovered, and inserted into the *map* and *gmap* modules, respectively. The list of *done* entries is read, and these entries are inserted into *done*, using operations of the *done* module. The guardian's stable state is read, and the stable objects are reconstructed.

After the *recover* routines have completed, the *map* startup routine is run. This procedure obtains a valid map from the map service, by calling the service's *insert* operation, with the guardian's *gid*, current crash count and current timestamp as arguments. The *gmap* module then obtains a new generation map and timestamp from the generation service, in the same manner. Once the stable state of the guardian has been reconstructed, and a valid map, gmap and timestamps have been obtained, the guardian may resume normal activity.

When a guardian is destroyed, its crash count and generation count are set to infinity. This is accomplished using the *delete* operations of the *map_service* and *gen_service* modules. There are two different ways in which a guardian can be destroyed: it can destroy itself, by executing the Argus *terminate* statement, or it can be killed by the guardian manager. If a guardian executes a *terminate* statement, it calls the map and generation services, to delete its own *gid*. If the guardian manager kills a guardian, it is responsible for notifying the service.

In order to ensure that the map and generation services actually receive the information about a destroyed guardian, a guardian that executes a *terminate* statement is not allowed to disappear until the services' *delete* operations have completed. However, it is possible for a guardian to crash while the delete operations are still running. In this case, the guardian must recover from the crash, and then attempt to delete itself once more. For this reason, before a guardian notifies the service of its impending destruction, it records on stable storage the fact that it is trying to terminate itself. Upon recovery from a crash,

the guardian checks to see if it was previously trying to destroy itself. If termination was in progress at the time of the crash, the guardian does not accept any handler calls after recovery; it simply tries to delete itself from the service once more.

The above deletion procedure works correctly for guardians, but not for Argus *programs*. As explained in Section 2.1, *programs* are Argus modules that do not have stable storage and are not resilient to crashes. If a program ever crashes, because of a node crash or software failure, it is permanently destroyed, but it cannot recover to notify the map and generation services. To ensure that destroyed programs are eventually deleted from the service, the guardian manager could periodically send the service a list of currently active programs and guardians at the manager's node; any map or gmap entry from the manager's node, but not in the manager's list, would indicate a destroyed program that should be deleted from the service. Implementation of this method would require that the guardian manager be able to determine which programs are still running. For example, programs could contact the manager at the time they were created; the manager could periodically check to see if a program's process was still active. Alternatively, if the manager only performed this function during recovery from a node crash, it could simply report that no programs were active at its node, because all of them were destroyed by the crash.

It should be noted that the guardian manager was not actually altered as a part of this thesis work. Changes to the manager were avoided in order to avoid interfering with any other users of Argus. Because the manager is only involved when guardians are created and destroyed, not while they are running normally, the decision not to re-implement the manager should not significantly alter the performance results we obtained.

6.3 Handler Call Processing

When an action makes a handler call, the following steps are performed. First a new action is created, with the current action as its parent. This new action is known as the *call action*. Next, a call message is constructed and sent to the guardian whose handler is being

invoked. The call message includes the *gid* of the recipient, the name of the handler, the arguments to the call, and additional information used for action management, including orphan detection. This information includes the *aid* of the sender, the sender's *dmap* (i.e. the list of guardians the sender is dependent on, and their crash counts), the sending guardian's map timestamp, generation timestamp, and the guardian's copy of *done*.

After the call message is sent, the sending action waits for the reply. If a reply is not received within a reasonable time (15 seconds), the *rpc* module tries to contact the recipient, to verify that the handler call is still active. If the recipient cannot be contacted, then the handler call is aborted, and an *unavailable* exception is signalled to the caller. If the recipient can be contacted, but the call is not active there, then the call action is aborted and the call is retried. Otherwise, the caller waits a longer period of time (1–4 minutes), and contacts the recipient again if a reply has not been received.

When a guardian receives a call message, it reads the caller's *aid*, and creates a new action to run the call. This new action is called the *handler* action, and is a child of the call action. After the handler action is created, orphan detection information is read from the call message. First, the caller's *dmap* is extracted. Next, the map timestamp, generation timestamp and *done* entries are read, using operations in the *map*, *gmap* and *done* modules. If necessary, the *map* and *gmap* routines obtain a new *map* or *gmap*, respectively, from the map and generation services. The routines used to read the timestamps and *done* entries also kill any local orphans they can detect. Next, the recipient determines whether the calling action is an orphan. A *map* routine compares the entries in the caller's *dmap* with the recipient's map values for the same guardians. If any guardian listed in the caller's *dmap* has a larger crash count in the recipient's map, then the caller is a crash orphan. In this case, a *refusal* message is sent to the caller, and the handler action is aborted; otherwise the handler call is allowed to run normally.

After orphan detection is performed, if the message is not refused, the arguments to the call are extracted from the message and the call is run. When the call completes, the

results of the call are returned to the caller, along with the replying guardian's map and generation timestamps, and a list of its *done* entries. The handler action is then committed or aborted, as appropriate.

When the calling action receives the reply message, it extracts orphan detection information from the reply. First, the replying action's *dmap* is read. Next, the replying guardian's generation timestamp, *done* entries, and map timestamp are read and used to destroy local orphans. Finally, the results of the call are read, the call action is committed, and the results are returned.

It is possible for the call action itself to be detected as a orphan when the reply message is received. For example, suppose the call action C has topaction ancestor A and intermediate ancestor B. B could abort, and then A could commit. Alternatively, A itself might abort. In either case, A's guardian might advance its generation count and notify the generation service that A's generation has completed. When C receives the reply from its handler call, it might need to contact the service to get a new *gmap*, because the reply message might have a larger generation timestamp than the caller's guardian has. When the new *gmap* is received from the server, C will be detected as an orphan. Similarly, it could be determined that C is a crash orphan, based on information obtained from the map service. In either case the call must be aborted, and the caller must be destroyed.

It is also possible for the replying action to be an orphan even if the caller is not. For example, the replying action may have become dependent on a guardian that subsequently crashed; the caller may have learned of the crash from the map service. This condition is detected by comparing the replying action's *dmap* with the calling guardian's map. In this case, the reply is refused, the call action is aborted, and the call is retried.

If the call action receives a *refusal* message from the recipient guardian, then either the handler action running at the recipient was destroyed as an orphan, or the call action itself was rejected as an orphan. In either case, the call action is aborted. The *done* entries and timestamps in the refusal message are then used to determine whether the caller is an

orphan. If the caller is, in fact, an orphan, then it is destroyed; otherwise, the call is retried.

6.4 Dealing With Crash Orphans

The Argus system uses an abstract data type called *map* to detect and destroy crash orphans. This data type provides operations to read map information from a message, write map information into a message, return the map value (i.e. crash count) for a guardian, return the current map timestamp, and merge an incoming map with the current map. Also provided are a start up operation to obtain an initial map from the map service, an operation that returns all of the entries in the map, and a routine to set the timestamp to the previously saved value during recovery from a crash.

In the basic Argus implementation, *map* operations were used to read and write complete maps to and from messages. In our optimized scheme, these same routines are used to read and write timestamps, and to call the map service if necessary. In this way, both our use of the service and our sending of timestamps in messages, instead of whole maps, are hidden from higher level modules. This minimizes the changes needed for these modules and allows any future optimizations of the orphan detection algorithm to be implemented more easily.

The representation of the *map* consists of three elements: the timestamp of the current map, a list of map entries, and a flag indicating whether the map timestamp has changed since the last time it was written to stable storage. The list of map entries is implemented as an array, sorted by *gid*. Each entry in the list is of the form (gid, crash count) or (gid, crash count, timestamp), where gid is the id of some guardian, crash count is our crash count value for that guardian, and timestamp is a stamp indicating when we received the entry. Timestamp values are maintained only for deleted entries (i.e. entries with crash counts of infinity, representing destroyed guardians), and are used for garbage collection, as explained below.

In order to prevent *map* operations from seeing the representation in an inconsistent state, some form of synchronization is necessary. In the *map* implementation, synchroniza-

tion is provided by *critical sections*, during which other threads within the guardian are not permitted to run. This is also known as *disabling interrupts*, because execution of the current thread cannot be interrupted to allow another thread to run. However, even if the current thread is in a critical section, other threads may run, if the current thread suspends its execution to wait for some event.

The implementation of many of the *map* operations is fairly straightforward. For example, the *lookup* routine simply searches the map array to find the entry for a *gid* and then returns its crash count value. The *put* procedure writes the current timestamp into a message, by calling the *put* operation of the the timestamp data type. The *startup* procedure is called when the guardian is first created and when it recovers from a crash. It obtains a new map from the map service, by calling the service's *insert* operation. The arguments to *insert* are the guardian's *gid*, its crash count, and either a zero timestamp, if the guardian has just been created, or the saved timestamp if the guardian is recovering from a crash.

The operations performed when map information is read from a message are more involved, and will be discussed at greater length. The *map* data type provides a *get* procedure to extract and process map information from an incoming message. When this routine is called, it first reads the map timestamp contained in the message. Next, the incoming timestamp is compared with the recipient's stamp. Unless the incoming stamp is less than or equal to the guardian's own stamp, the map service's *refresh* operation is called, with the incoming stamp as an argument. Once a map has been obtained from the service and merged with the existing map, the *get* routine returns to its caller.

While the *refresh* operation is in progress, execution of the current thread is suspended, and other threads are permitted to run; this allows the guardian to continue processing other calls. The *refresh* request is processed by a separate thread within the *map_service* data type; this thread restarts the requesting thread when the *refresh* is completed.

Although the map and timestamp returned by *refresh* could be merged with the current map and stamp within the *get* operation, in our implementation this is done by the

map_service itself. Whenever the *map_service* receives a new map and timestamp from a replica, it calls the *merge_map* operation of the *map* data type, with the new map and timestamp as arguments. *Merge_map* destroys any orphans that can be detected using the new map, and then updates the current map and stamp.

Calling the *merge_map* procedure from within the *map_service* violates the abstraction barrier between client and service. However, it eliminates some duplicated work. By merging new map versions directly into the client's map, the *map_service* can avoid caching its own copy of the map. Instead of processing a new map twice (once to merge it when it is received from a replica; a second time when the client uses it to kill orphans), we process the map only once, using a single operation.

To make our discussion of the *merge_map* procedure clearer, we will give names to the different map versions and timestamps handled by this routine. Let s_r be the map version received from a replica, and t_r be its timestamp. The guardian's original map, before the new map was received, will be named s_o and its timestamp will be t_o. The result of merging these two maps will be a new map s_n, with timestamp t_n. As in our discussion of the map service, $s(G)$ will refer to the value of G in s, and $t[i]$ will be the ith component of t.

When *merge_map* is invoked, it first compares the guardian's current timestamp t_o with the received stamp t_r. If $t_r \leq t_o$, then *merge_map* returns immediately, without altering the map and stamp. Otherwise, *merge_map* compares s_r and s_o, to detect local orphans and construct the new map s_n. Because s_r and s_o are implemented as sorted arrays, it is easy to scan their entries in order, searching for differences between them. After all orphans have been killed, s_n is installed as the guardian's map.

If a guardian id G is found that is present in both maps, with a larger crash count in s_r, then any local actions dependent on that guardian are crash orphans, and are destroyed. An entry for G is then added to s_n with the crash count specified by s_r. If both maps contain an entry for G, but $s_r(G) \leq s_o(G)$, then G's entry from s_o is included in s_n.

If s_o does not contain an entry for G and s_r does, then the entry from s_r is added to s_n. No orphans can be present in this case, because either $s_o(G) = -\infty$, in which case no actions dependent on G ever ran here, or $s_o(G) = +\infty$ (i.e. G has been garbage collected from s_o), in which case all such actions have already been destroyed. If G has been garbage collected from s_o and $s_r(G) < \infty$, then G's entry from s_r must not be added to s_n; we will show later that this case cannot arise. If $s_r(G) = +\infty$, it is possible that G has already been garbage collected from s_o; however we have no way to determine whether this is true. In this case, we add an entry for G to s_n, with a value of $+\infty$; the entry is eventually garbage collected again.

If G is present in s_o, but is missing from s_r, there are two possibilities: either s_r is an old state that does not reflect the fact that G was inserted into the map service, in which case $s_r(G) = -\infty$, or G has been deleted from the service and then garbage collected, so $s_r(G) = +\infty$. If $s_r(G) = -\infty$, then G's entry from s_o is included in s_n. If $s_r(G) = +\infty$, then G has been destroyed, and any local actions dependent on this guardian are killed; in this case an entry for G is not included in s_n (i.e. G is garbage collected).

The correct value for a guardian id G that is missing from s_r can be determined by comparing timestamps. If $t_r \geq t_o$, then $s_r(G) \geq s_o(G)$. We know that $s_o(G) > -\infty$, because values of $-\infty$ are never inserted into the map. Therefore, the new value for G must be $+\infty$. If t_r and t_o are incomparable, then $s_r(G)$ must be $-\infty$; the reason for this is given below. It is not possible that $t_r \leq t_o$, because *merge_map* returns immediately in that case, without examining the new map.

It can easily be shown that if t_r and t_o are incomparable, and $s_o(G)$ is finite and G is missing from s_r, then $s_r(G)$ is $-\infty$, not $+\infty$. The entry for a deleted gid G cannot be garbage collected from the map of a replica R until R is sure that all other replicas have learned about G's deletion. We will show that at this point R's timestamp will be greater than or equal to t_o, i.e. the stamps will be comparable. Because t_r and t_o are actually incomparable, however, G cannot have been gargage collected from s_r. Since G is

missing from s_r but has not been garbage collected, The only remaining possibility is that $s_r(G) = -\infty$.

Now let us show that if G has been garbage collected by R then $t_r \geq t_o$. This is equivalent to showing that $\forall j, t_r[j] \geq t_o[j]$, i.e. each component of t_r is greater than or equal to the corresponding component of t_o. Let J be the replica that is allowed to increment the jth component of a timestamp. Let X be the value of the jth component of J's timestamp immediately after J learns about the deletion of G. R cannot garbage collect G from its state until it receives gossip from J, indicating that J has learned of the deletion. At this point the jth component of R's stamp is greater than or equal to X. A timestamp whose jth component is larger than X can only be created after the deletion of G. Such a stamp must either be created at J after the deletion, or derived from such a stamp by merging. In either case, the stamp refers to a state in which G's value is $+\infty$. Because the client's state s_o has a finite value for G, the jth component of its stamp t_o must be less than or equal to X. Therefore, $t_o[j] \leq X \leq t_r[j]$; this is true for every j. Therefore, $t_r \geq t_o$.

We will now examine the method by which *merge_map* garbage collects deleted entries from the client's map. The procedure for garbage collecting entries must be considered carefully. A deleted entry must not be garbage collected from the map if there is any chance that we may later add a finite value for the same *gid* by mistake. We need to be sure that if we ever receive a map from the server with a finite value for the deleted *gid*, its timestamp will be less than ours. In that case, *merge_map* will not alter our map at all.

If we receive a map s_r from which a guardian id G has been garbage collected, its stamp t_r (and our stamp t_n after we merge t_r and t_o) will be larger than the stamp of any state with a finite value for G, as shown above. If we later receive a state with a finite value for G, its stamp will be less than ours, and *merge_map* will discard it. Thus, we can safely garbage collect the entry for G. However, there is one case in which we cannot tell whether G has been garbage collected from s_r. If s_o contains a value of $+\infty$ for G that has not been garbage collected, and G is missing from s_r, and t_r is not comparable to t_o, we cannot

tell whether $s_r(G)$ is $+\infty$ or $-\infty$. (Our earlier argument that $s_r(G) = -\infty$ when the stamps are incomparable applies only when $s_o(G)$ is finite.) One solution to this problem is simply to wait until we receive a state in which G is missing and for which $t_r \geq t_o$, and then garbage collect G. However, because we communicate with different replicas, which may have incomparable timestamps, and because our stamp increases when we receive a new state, it could be a long time before we receive a stamp greater than our own. Thus, garbage collection could be delayed indefinitely.

We solve this problem by maintaining a timestamp for each deleted entry. The timestamp t_G of an entry $\langle G, \infty, t_G \rangle$ is the stamp of the first state we received that contained an infinite value for G. When we receive a state s_r from the server such that $t_r \geq t_G$, we know that $s_r(G) = \infty$. If G is missing from s_r, then it must have been garbage collected by the server, and we can safely remove it from our own map. Sooner or later, because of exchanges of gossip messages, a replica will be able to purge its entry for G and advance its timestamp to be greater than or equal to t_G, although not necessarily greater than our current stamp t_o, which is also increasing. We should eventually receive from the server some state with a timestamp greater than t_G; the entry for G can then be garbage collected safely. If gossip messages are sent reasonably frequently, are delivered successfully most of the time, we should not have to wait very long for this to happen.

It should be noted that our treatment of entries missing from s_r and our method of garbage collecting deleted entries rely on the implementation of the server replicas. This violates the abstraction between the client and the map service. However, if we think of the *merge_map* routine as a part of the *map_service*, rather than part of the client, this violation disappears. It seems reasonable for the map service front end to know about the way in which replicas handle garbage collection of deleted entries.

When the *merge_map* routine discovers an orphan, that orphan is destroyed. This is accomplished by setting a flag in the orphan's state information; the orphan's thread is then allowed to run and destroy itself. *Merge_map* waits until the orphan is gone before

resuming execution.

While *merge_map* is waiting for an orphan to destroy itself, other threads may also run, and new orphans may be created. Some of these new orphans may be dependent on guardians that *merge_map* checked earlier. We cannot prevent new orphans from forming while old ones are being destroyed, because we cannot add entries to the current map one at a time. If entries were added one at a time, the partially constructed map would not have a valid timestamp. If other threads within the guardian were allowed to run while the new map was only partially constructed, they could see the map and timestamp in an inconsistent state.

Because new orphans may be present, and may be dependent on guardians examined earlier, *merge_map* must scan the new map as many times as necessary, until no further orphans are found. If no orphans are present, the map only has to be scanned once. If orphans exist, but no new orphans are created while we wait for the old ones to be destroyed, the map will be scanned twice. Orphan detection should usually be completed relatively quickly, so new orphans should not normally be present. Eventually (usually the first or second time), we should find no orphans; however, there is no strict upper bound on the time required. Such an upper bound could be imposed by prohibiting new handler calls from starting while orphan detection from a previous call was in progress. We do not believe that many new orphans should form in the short time normally needed for orphan detection; delaying new calls during this time seems unnecessary.

After all orphans have been killed and s_n has been constructed, s_n is installed as the guardian's map. The timestamps t_o and t_r are merged together to obtain the new timestamp t_n, which is then installed as the current stamp. A flag is set in the *map* representation, indicating that the map timestamp has changed. The next time the guardian performs two-phase commit, the *recover* module will save the new timestamp on stable storage, and reset this flag.

One question that has been overlooked so far is the actual method for constructing s_n

during the initial comparison of s_o and s_r. One possibilty is to implement s_n as an array containing the elements of the new map; each time we discovered an entry that belongs in s_n we would add it to the array. However, it is more efficient to construct a list Δs of differences between s_n and s_o rather than constructing s_n itself. Each entry in this list is either a *new_gid* entry, a *changed_value* entry or a *garbage_collect* entry. While comparing s_r and s_o, if we find a new gid G that should be added to the map, a *new_gid* entry for G is added to Δs. This entry contains G's gid and crash count. When a gid G is found that is present in s_o and that should have a new crash count in s_n, a *changed_value* entry for G is added to Δs. This entry contains G's gid, its new crash count, and the array index at which G's entry is located in the representation of s_o. If G has the same value in s_o and s_n then no entry for it is added. If G should be garbage collected from the map, then a *garbage_collect* entry containing its gid is added to Δs.

After the initial comparison of s_o and s_r, and after any further passes to kill orphans, if necessary, *merge_map* uses Δs to construct s_n from s_o. If there are no differences between s_o and s_n (i.e. Δs is empty), then the map is not modified. If Δs contains only *changed_value* entries, then the corresponding entries in s_o are modified to have the appropriate new values; this can be done quickly because each entry in Δs contains the array index of the corresponding entry in s_o. If new gids were added or old gids were garbage collected, then a second pass is performed to merge Δs with s_o. This can be done in linear time, because Δs is already sorted by gid. No other threads are allowed to run while the map is being modified; this prevents the other threads from seeing an inconsistent map and stamp.

Our method for constructing the new map minimizes the work required when there are no changes to s_o, or when the crash counts of gids have changed but no gids have been added or garbage collected. We believe that the latter will be the most common case. We have found that this implementation allows *merge_map* to run in half the time required for the alternative implementation in which the complete s_n is constructed and then installed.

While comparing maps and destroying orphans, the *merge_map* routine constructs a

list of local roots of the orphans that have been found. Only local roots that are handler actions are included in the list; topaction local roots are omitted. After the maps and timestamps have been merged, refusal messages are sent to the guardians from which the calls were made, to allow the guardians to abort the calls. We do not try hard to deliver these messages; as explained above, the caller will eventually discover that the call is no longer active and take appropriate action.

6.5 Abort Orphans

Two abstract date types, *done* and *gmap*, are used to deal with abort orphans. *Done* holds the list of aborted actions this guardian knows about; *gmap* stores the generation map.

6.5.1 Implementation of Done

The *done* module provides operations to add a new entry to done, determine whether an action is an abort orphan, write done into a message, read done entries from a message, and kill abort orphans. There are also operations to yield all entries of done, or those not yet saved on stable storage, insert entries recovered from stable storage after a crash, and a routine to delete old entries based on information in the generation map.

The representation of *done* consists of a list of entries, and a counter indicating the number of entries not yet saved on stable storage. Each entry consists of an *aid* for an aborted action and a flag indicating whether that entry has been saved. The list of entries is implemented using an array, sorted by *aid*. Duplicate entries are never inserted into the list. Whenever an *aid* is inserted, the *aids* of its descendants, if any, are removed, because they are redundant.

The *aid* data type provides a "less than" operation which makes this sorted representation possible. It defines a total order on *aids*, such that the id of each action is less than

the *aids* of its descendants. In addition, every action B not descended from an action A is either less than A or greater than all of A's descendants.

We use critical sections to synchronize access to the *done* representation, just as we did for the *map*. Other threads within the guardian are not allowed to run while the representation is being scanned or modified. However, when an orphan is found, its thread is allowed to run so it can destroy itself; it is possible for other threads to run at this time as well.

The *put* operation of the *done* data type adds the *aids* in *done* to a message. The *aids* are added in ascending order, to facilitate merging of incoming and existing *dones* when a message is received.

The *orphan* operation is used to determine whether an action A is an abort orphan. This operation takes the action's *aid* as an argument, and returns a boolean value indicating whether it is an orphan. As mentioned in Chapter 3, we define the *origin* of an action as the guardian at which the action's non-nested topaction ancestor was created. The *orphan* routine first compares the action A's generation number with the generation count of its origin; the origin's generation count is obtained from the *gmap* module. If A's generation number is lower than the generation count of its origin, then A must be an orphan, because all top-level actions of that generation have completed. Otherwise, the list of *done* entries is searched, to determine whether it contains an ancestor of A. If so, A is an abort orphan; otherwise, A is not.

An *update* operation is provided to add the *aid* of an aborted action A to *done*. This routine also kills any local descendents of A, because these descendants are now orphans. It is important that the entry for A is not added to *done* until all of A's descendants have been destroyed. Otherwise, if the guardian received a handler call from an action that was already aware of the abort, the call would be allowed to run; the guardian would see that A's id was already in *done* and assume that there were no active orphans. The orphans and the new call could then interfere with each other. For example, the new call could modify

an object for which A previously held a lock; the orphans would assume that A still held the lock, and could behave incorrectly when they observed the modifications.

When the *update* procedure is invoked, it first compares the generation number of A with the generation count of A's origin, as listed in *gmap*. If A's generation is smaller, then A's descendants can be detected as orphans using the information in *gmap*, so there is no need to add A to *done*. If A does not belong to an old generation, the list of *done* entries is searched to determine whether an ancestor of A is already in *done*. If A has an ancestor in *done*, then A is not added, because its descendants can already be detected as orphans. If it is necessary to add A to *done*, then any local actions descended from A are destroyed as orphans. Next, A is added to the list of entries, and any entries for descendants of A are deleted.

The sorted array representation of *done* makes it easier to find the ancestors and descendants of A. To find an ancestor of A, we search for the largest *aid* that is smaller than A; this is the only entry that can be one of A's ancestors. This search also leads us to the array position P into which A should be inserted, if no ancestor is present. Any *done* entries for actions descended from A will occupy consecutive locations in the array, beginning at postion P. These properties are ensured by the specification of the "less than" operation on *aids*. For any two actions B and C, either C is less than B, a descendant of B, or greater than all of B's descendants; these possiblities are mutually exclusive. If an *aid* A has an ancestor B in *done*, then any *aid* greater than B and less than A must be a descendant of B, and cannot be listed in *done*. Therefore, B must be the largest *aid* in *done* that is smaller than A. All actions not descended from A are either smaller than A or larger than A's descendants; therefore A's descendants have the smallest *aids* that are larger than A.

As mentioned in our discussion of the map, orphans are not destroyed directly. Instead a flag is set in the orphan's state information; the orphan is then allowed to destroy itself. If the *update* routine finds any orphans and waits for them to be destroyed, it is possible for the *done* array to be altered or for new orphans to be created. Therefore, after destroying

the first set of orphans, we search the done array again, find the position into which to insert A, and check for ancestors and orphans. Once we determine that no orphans are present, a *done* entry for A is added, and any entries for descendants of A are deleted.

While searching for and destroying orphans, the *update* routine keeps track of any handler actions that have been destroyed. After all orphans have been killed and *done* has been updated, a process is forked to send refusal messages to the callers of any handler actions that were destroyed as orphans.

The *done* data type also provides a *get* operation, which performs the work needed to extract *done* information from a message, merge it into our own *done* and kill any local abort orphans. First, this routine calls the *gmap* module's *get* operation, to extract the generation timestamp from the message and update the generation map; the *gmap* module also kills any orphans it can find. Next, the *aids* in the incoming *done* are read. As each *aid* A is read, *get* compares its generation number with the generation count of its origin, as listed in *gmap*. If A's generation number is smaller, then its descendants can already be detected as orphans using *gmap*, and any such orphans have already been destroyed. In this case, A is not added to *done*. Otherwise, all descendants of A are destroyed as orphans, A is added to *done*, and any entries for its aborted descendants are removed from *done*. The procedure followed in this case is similar to the *update* operation. Finally, for each handler action destroyed as an orphan, a refusal message is sent to the caller.

It is easy to find the position into which to insert each new *aid*, to keep the *done* array sorted. Because both the old and incoming *done* are sorted, we can scan the old done to find the place to insert one new *aid*; the search for the place to add the next *aid* can start at this position, rather than at the beginning of the array. As in the case of *update*, however, we must start our search over from the beginning of the array if we had to wait while orphans were destroyed.

The *done* module includes a *timer* procedure to delete old entries from the *done* list. This procedure searches the list for *aids* whose generation numbers are lower then the generation

counts of their origins. Each such action is removed from *done*. However, some of these entries may already have been saved on stable storage. Periodically, the stable storage log is compacted by the *recover* module; at this time the deleted entries are removed from the stable version of *done*.

During crash recovery, *gmap* is used to eliminate any old *aids* that may be present in the stable copy of *done*. When the guardian recovers from a crash, the generation number of each *aid* in *done* is compared with the *gmap* value of its origin; *aids* with old generations are not be added to the volatile *done*. The entries for these *aids* are purged from stable storage the next time compaction is performed.

In our implementation, the *timer* routine is called by the *gmap* module, when a new generation map is received from the generation service. This seems like the best time to remove deleted entries; it keeps *done* as small as possible, and prevents us from wasting time trying to garbage collect *done* when the *gmap* has not changed. Alternatively, *done* could check the generation timestamp periodically (or after each call to *gmap*), and call the timer if the generation stamp has changed.

6.5.2 Implementation of the Generation Map

The generation map is implemented by the *gmap* data type. This data type provides operations to read the generation timestamp from a message, write the timestamp into a message, return the generation count of a guardian, return the current generation timestamp, and merge a *gmap* from the generation service with the current generation map. The implementation of *gmap* is very similar to that of *map*. The *gmap* representation includes an array of entries, sorted by gid, the current timestamp, and a flag indicating whether the current stamp has been saved to stable storage. Generation maps are merged and orphans are destroyed in essentially the same manner as for *map*. *Gmap* recognizes an action as an orphan if the action's generation number is lower than the new generation count for its origin, as listed in the new *gmap* obtained from the service.

One question that arises is how often to advance the generation counter. Advancing the counter more frequently allows finer control over how much information accumulates in *done*. The only cost of advancing the generation count is the time needed to write the new count to stable storage. One solution is to advance the counter each time a new (non-nested) topaction is created, but to save this counter only every Nth generation. Upon recovery from a crash, the guardian sets its generation counter to the next larger multiple of N. In our implementation, the counter is advanced after each topaction is created, but saved after every 256 topactions.

If a guardian remains in existence long enough, its generation count may eventually wrap around, producing smaller (i.e. negative) generations for later actions. Our algorithm will not work correctly if generation wrap occurs; therefore the counter must be made large enough, or the rate of generation advancement small enough, that wrap will not occur.

A process within the *gmap* implementation is responsible for notifying the generation service of generation changes. Periodically, this process examines the list of active actions originating at this guardian, to determine the earliest generation n to which any of them belongs. All topactions from earlier generations must have completed (committed or aborted), so the generation service is notified of this fact, i.e. the guardian's *gid* is inserted into the service, with a value of n. If the guardian's current *gmap* value is already equal to n, then there is no need to notify the server.

6.6 Implementing Action Identifiers

Our implementation of action identifiers is essentially the same as in the basic Argus implementation, except for the addition of the generation number to the *aid* representation. Action identifiers are implemented as sequences of integers. Each *aid*'s representation is of the form (nested, generation, pairs), where *nested* is the number of nested topactions that are ancestors of this action, *gen* is the number of the generation in which the action's non-nested topaction ancestor was created, and *pairs* is a sequence of pairs of integers identifying

this specific action. A non-nested topaction has only one pair in its *aid*. A child action's id is derived from its parent's by adding an additional pair to the sequence. This makes easy to tell whether one action is an ancestor of another. The first integer in the pair is the *gid* of the guardian at which the action was created; the second is a sequence number identifying the specific action. The *ainfo* module keeps track of the number of children each action has, and supplies the correct sequence number to the *aid* creation routine when a new *aid* is needed. The negative of the *gid* is used instead of the gid if the action is a topaction. It is necessary to indicate nested topactions in some way, in order to recognize that two-phase commit should be performed for them.

The *aid* data type provides operations to create *aids* for topactions, children, concurrent siblings and handler actions, determine whether a given *aid* is the id for a topaction, return the non-nested topaction ancestor for an *aid*, test whether one action is an ancestor of another, and to read *aids* from and write them to messages. Also provided is the "less than" operation described above. The implemenation of most of these operations is fairly straightforward.

One question that arises is how to implement the "less than" operation. This routine defines a total order on *aids* such that ancestors' ids are less than their descendants', and each action not descended from an action A will have an *aid* less than A's or greater than the *aids* of A's descendants. The desired properties can be obtained by comparing the elements of the sequence of pairs contained in each *aid*. When *aids* A and B are compared, the first pair in A's sequence is compared with first one in B's sequence; if they are the same, the second pairs in the sequences are compared, and so forth, until a difference is found. At this point, the *aid* whose pair has the smaller *gid* is considered smaller; if the *gids* are equal, the *aid* whose pair has the smaller sequence number is smaller. If the *aids* are of unequal length, and no differences are found, then the shorter id is an ancestor of the longer one, and is therefore considered "less than" the longer id. If an action B is descended from A, its *aid* will differ from A's beginning at position $n+1$, where n is the length of A's id. Every action not descended from A will differ from A at an earlier position, and will

therefore be either less than A or greater than all of A's descendants.

6.7 How to Abort an Orphan

Normally, aborting an orphan is fairly simple. First, a flag in the orphan's state information is set, indicating that someone is trying to abort it. If the orphan's thread is blocked waiting for some event, the thread is made runnable. When the thread wakes up, it checks whether anyone is trying to abort it. If the thread discovers that it is being aborted, it proceeds to destroy itself as follows. First, all local descendants of the orphan are aborted. If any of these descendants are call actions, the *aids* of their handler actions are added to *done*. The *update* operation of the *done* module is used to accomplish this. Next, any locks the orphan holds for atomic objects are broken, its versions of atomic objects are discarded, and the previous versions are restored.

There are three complications that can arise when an orphan is aborted. First, the orphan may hold mutexes for some objects. Because mutexes are commonly used to protect an object temporarily, while it is in an inconsistent state, destroying an action while it holds a mutex may leave these objects in an inconsistent state. Therefore, if an action holds a mutex, it does not destroy itself when it wakes up, even if someone is trying to abort it. Instead, it keeps running normally until its last mutex is released; it then proceeds to abort itself. It is possible for an action to run indefinitely without releasing all of its mutexes; if this happens, the only way to abort the action is to crash the guardian where it is running.

It should be noted that, when an action receives a *refusal* message in reply to a handler call, because it has been detected as an orphan by the recipient of the call, it normally destroys itself. However, if the action holds a mutex, it cannot destroy itself; nor can it continue running until the mutex is released, because it has not received a normal reply to the handler call. In this case, the handler call signals *unavailable*, to allow the action to keep running, and eventually release its mutex and be destroyed.

A second problem occurs when the action being aborted is not a local root. In this case,

the action's parent expects it to satisfy some specification; if the action is destroyed as an orphan, at some arbitrary point, it may fail to meet its specification. Because non-atomic data and global variables may be shared between the action and its parent, these data may be left in an inconsistent state when the action is destroyed. In this case, the parent is said to be *stranded*. In order to avoid this problem, whenever an action is destroyed as an orphan, its local root is also aborted.

The third problem concerns orphans that are non-nested topactions, or whose local roots are non-nested topactions. If the orphan is aborted, its local root is aborted too. However, the local root topaction is a part of some thread which expects it to meet a specification. Aborting the root would leave the thread *stranded*. In this case, we must wait for the orphan to terminate on its own. If the orphan does not terminate within a reasonable time, it may be necessary to destroy it by crashing the guardian.

Related to the topic of destroying orphans is the issue of unwanted committed subactions. These are committed local roots which have not yet committed through the top, and which are discovered to have aborted ancestors or to depend on a guardian which has crashed. Technically, these actions are not considered orphans, because they are no longer active. However, because such actions occupy space, may hold locks, and can never succeed in committing through the top, it is desirable to destroy them.

Unwanted committed subactions can be destroyed when a message is received containing a piggybacked *done*, or when a new map or generation map is received from the map or generation services. A committed local root can be destroyed if it has an ancestor in *done*, has a generation number smaller then the generation count of its origin, as listed in *gmap*, or if a guardian in its *dlist* has crashed, as indicated by the information in *map*. The *committed* data type, which holds the list of committed actions, provides operations to destroy all committed local roots that are descended from a given action, or depend on a given guardian, or that belong to old generations. In each case, the action is removed from the committed list, its locks are broken and its versions of atomic objects are discarded.

Committed local roots cannot hold mutexes, so there is no need to be concerned with that case.

6.8 Generation Extension

Our scheme for using the generation map to determine when information can be removed from *done* may perform poorly if some actions take a long time to complete. Until all actions of a given generation have completed, neither that generation nor any subsequent one can be inserted into the generation service. Guardians must retain *done* entries from that generation, and all subsequent ones, until the actions terminate and the generation count in *gmap* is advanced. Therefore, if a generation contains even one long action, garbage collection of *done* entries from that generation, and all subsequent ones, may be delayed substantially. One solution to this problem is to increase the generation numbers of old actions, if necessary, to allow the generation count to be advanced. Once the generation numbers of all actions from old generations have been advanced, a larger generation count can be inserted into the service. The guardian's *gmap* value can be increased, and garbage collection of *done* entries becomes possible. We have developed a method for advancing generation numbers of actions, as discussed below; however this scheme has not yet been implemented.

Our method of advancing generation numbers is based on the deadline extension algorithm discussed in [12] and [14]. By analogy with extension of deadlines, we call our scheme *generation extension*. When generation extension is used, we no longer (physically) include the generation number of an action within its *aid*. Instead, each guardian maintains a generation table listing active actions and their generation numbers. Committed actions that have not yet committed through the top are also included. Only local root actions need to be included in this table; the generation number of any other action is the same as the generation of its local root ancestor. An entry in *done* for an *aid* A also includes A's generation number as of the time A was inserted into *done*.

The existence of the generation table could be hidden from other parts of the system, by making the table a part of the *aid* data type implementation. In this case, *aid*s could be manipulated using the same operations as before; however, the *get_generation* operation on *aid*s would now perform a table lookup, instead of fetching a generation number contained physically within the *aid*.

A generation table entry has the form (aid, generation, extension_status). *Aid* is the id for this action; *generation* is its generation number. During generation extension, the extension status for an action A consists of an array containing entries for A's active descendants. This array contains one entry for each active handler call whose call action C is a "purely local" descendant of A (i.e. there are no other calls "between" A and C). Each of these entries is of the form (id, acked, time), where *id* is the *aid* of the handler action for the call, *acked* is a flag indicating whether the target guardian for this call has finished extending the call's generation number, and *time* is the time at which we will send a new extension message if a reply has not yet been received. When generation extension is not in progress for A, its extension status is set to the special value *not_extending*.

Two rules must be observed to ensure the correctness of generation extension. First, the generation of a child must never be greater than the generation of its parent. This ensures that, if the parent aborts, the child can be detected as an abort orphan after the parent's done entry has been purged, using the information in *gmap*. Second, the generation service must not be notified that a generation has completed until extension has been completed successfully for all top level actions of that generation and all previous generations, and for all of their non-orphan descendants. This prevents these actions from being mistakenly destroyed as orphans.

As before, the *gmap* module periodically checks to see which of the guardian's generations have active actions. If the first $n-1$ generations have completed, then the guardian's *gid* is inserted into the generation service, with a value of n. When the *gmap* module checks to see whether any old generations have completed, it may discover that some actions from

very old generations are present. In this case, the guardian may decide to advance the generation numbers of these actions, so that a larger generation count can be inserted into the generation service. If the guardian decides to advance its *gmap* value to n, then generation extension is started for each active top-level action that originated at the guardian and has a generation number that is less than n.

To extend the generation of an action A, the guardian first increases the A's generation number, as listed in the generation table. This new generation number is sent along with the *aid* of A in any message in which the *aid* is included. Any purely local descendants of A have their generation numbers increased automatically, because their generations are taken from the table. Next, A's purely local call action descendants are identified, and added to A's table entry. Generation extension messages of the form *advance_generation*(*aid*, *new_generation*) are sent to the descendants' guardians. In the extension message, *aid* is the identifier of the descendant for which the message is being sent, and *new_generation* is A's new generation number. After all of the messages have been sent, the sender waits to receive acknowledgements for them. When an acknowledgement is received, the *acked* flag for the corresponding descendant is set. If a message is not acknowledged within a reasonable time, a new extension message is sent. When generation extension has been performed for all of the active descendants of A, the status entry for A is set to *not_extending*. Once extension has been completed for all actions of a given generation, and for all actions of previous generations, the generation service is notified that these generations have completed. If an action aborts, commits or is destroyed as an orphan, it is removed from the extension table, and no further attempt is made to extend its generation. If a handler call aborts, the call action is removed from the caller's extension table entry.

When a guardian receives a generation extension message for an action D, it determines whether D is active, committed through its local root, or neither. In this last case, either the action aborted or it never ran at this guardian. If the action is aborted, any active descendants it may have must be orphans, whose generations we do not want to increase. If the action never ran, it has no descendants to perform extension for. Therefore, if the action

is not active or committed, an acknowledgement of the form *extended(aid, new_generation)* can immediately be sent to the guardian from which the extension message was received, to indicate that generation extension for D has been completed. If D is still active, then D's guardian follows a procedure similar to that used for the topaction A. D's purely local call action descendants are identified, and extension records for them are added to D's generation table entry. Generation extension messages are sent to each descendant's guardian, replies are waited for, and extension messages are sent again if necessary. When all replies have been received, an *extended(aid, new_generation)* message is sent back to the guardian of D's parent, i.e. the guardian from which the extension message for D was received. If extension is in progress for an action and a duplicate extension message (i.e. one with the same generation) is received, the duplicate is ignored. If extension has been completed already, then an an acknowledgement is sent back immediately. Extension messages containing generations less than an action's current generation are ignored as being too old; the topaction's guardian has already requested extension to a later generation, so the request cannot be current. Similarly, the generation number in acknowledgement messages allows a guardian to ignore old replies.

One issue that must be considered is whether to extend the generations of committed handler actions. Extending generations for these actions means that more extension messages have to be sent. It is not strictly necessary to extend the generations of committed actions; however, if their generations are not extended, then the generation map can no longer be used to destroy unwanted committed subactions. Since these actions are not active, there is no danger of their seeing inconsistent data, and they need not be destroyed promptly. If these actions hold locks, they will be destroyed when querying is conducted on behalf of other actions seeking to obtain the locks; they may also be destroyed when two phase commit is performed for their nearest topaction ancestor.

The above generation extension scheme should allow extension to be completed fairly quickly, provided that no guardians involved are crashed or inaccessible, and provided that the chain of handler calls is not too deep. Two optimizations can be made to improve the

performance of this algorithm. First, if several extension requests are being made to the same guardian at the same time (for different actions or multiple calls of the same action), they can be combined into a single message, reducing message overhead. Second, if the call chain is very long due to recursive handler calls, an algorithm similar to that given in [14] can be used to "short circuit" the recursion, allowing extension messges to be passed around the recursion loop just twice, regardless of the recursion depth. In [14], it is argued that recursion is the most likely cause of very long call chains; if this is true, then this approach seems promising.

The idea behind short circuiting is to extend the generations of all of the local descendants of a topaction T when an extension message is received. During the first pass around a recursive loop, a *done tag* for each descendant of T is set equal to the new generation number, but the generation number itself is not changed. If any of these actions aborts before extension is completed, the *done tag* is used in its *done* entry, instead of the generation number. During the second pass around the loop, the generation numbers themselves are increased. Each extension message includes the sequence of guardians that have propagated it; this makes it possible to determine when the message has travelled completely around a recursive loop. The algorithm insures that every action's *done* entry is tagged with a value at least as large as the generation number of the action's descendants. As a result, the *done* entry will not be removed from *done* until the action's descendants can be detected as orphans using *gmap*. Each extension message also includes a list of aborted descendants of the topaction, to prevent the generation numbers of orphans from being increased. For a more detailed description of short circuiting, the reader is referred to [14].

Chapter 7

Performance of the Orphan Detection Algorithm

In this chapter we examine the performance of our orphan detection algorithm. To provide a basis for evaluation, we compare the central server based algorithm described in this thesis with the deadlining scheme presented in [12, 14]. First, we briefly describe the deadlining optimization. Next, we examine the time each method requires to perform individual operations related to orphan detection. Finally, the behavior of the system as a whole is considered. The performance results presented for the deadlining method were obtained from [12].

7.1 Deadlining Optimization for Orphan Detection

An alternative method of optimizing the Argus orphan detection algorithm has been proposed by Edward Walker [14] and implemented by Thu Nguyen [12]. This method does not make use of a central server to hold orphan information. Instead, the full *map* and *done* are sent in each message between guardians, but *map* and *done* entries are deleted when they are no longer needed.

To allow *done* entries to be deleted, each action is assigned a *done deadline* at the time it is created. This deadline specifies a time at which the action will be destroyed if has not yet completed. When a subaction is created, it is assigned the same deadline as its parent. Because actions are destroyed when their deadlines expire, *done* entries for actions with expired deadlines can safely be discarded; any actions detectable as orphans using these

entries have already been destroyed. To allow these entries to be identified, each *aid* in *done* is tagged with the *done* deadline of the action to which it refers. When the current time at a guardian exceeds the time specified by this tag, plus the maximum clock skew, the entry may safely be deleted.

A similar scheme is used to delete old entries from *map*. Each action is assigned a *map deadline* when it is created. For a non-nested topaction, this deadline is equal to the current time plus a *map deadline period*, which is uniform for all guardians. Each subaction is assigned the same deadline as its parent. Actions are destroyed if their map deadlines expire. When a guardian recovers from a crash and adds its new crash count to its *map*, the map entry is tagged with a time value equal to the current time plus the map deadline period. Clearly, if any action became an orphan because of the crash, its non-nested topaction ancestor must have been created before the crash; therefore, the action will have a *map* deadline smaller than the tag of the map entry for the crash. When the current time at a guardian exceeds the tag value for a *map* entry, plus the maximum clock skew, the deadlines of any actions detectable as orphans using that entry must have expired, so these actions must have been destroyed. Therefore, the *map* entry may safely be deleted.

Walker [14] describes a protocol for extending the deadlines of actions, so that long actions can run to completion. However, because an action's deadline may expire before extension has completed, it is possible that some non-orphan actions will be destroyed by the algorithm. A discussion of the implementation and performance of the deadlining optimization for Argus orphan detection can be found in [12].

7.2 Basic Argus Operations

The Argus system currently runs on a group of MicroVAX-IIs, MicroVAX-IIIs and VAX-Station 2000s[1], running Berkeley UNIX (BSD 4.3 Version). These machines are connected by a 10 megabit Ethernet.

[1]Micro-VAX and VAX-Station are registered trademarks of Digital Equipment Corporation

To place our performance measurements in context, we first give the time needed to perform a few basic Argus operations in the absence of orphan detection. These measurements were taken with an empty *map* and *done*, and orphan detection was not performed. These times were taken from [7] and [12]. All times were measured on MicroVAX-IIs.

According to [12], the time needed to send a message to a guardian and receive a reply is 13.5 milliseconds. This represents the time needed to send a null low-level message, and receive a null reply. Handler calls take longer to execute, even in the absence of orphan detection, because of the additional information that must be transmitted in each call, and the extra processing that must be performed. A handler call that takes no arguments, does no work, and has no return value, can be executed in 18 msec. The time needed to execute a null topaction varies, depending on whether the topaction needs to write modified objects to stable storage. According to [7], a topaction that commits immediately, without making handler calls or acquiring locks can be executed in 0.63 msec. If the topaction acquires a read lock for some local object and then commits, 0.84 msec is required; if a write lock is acquired, then 17.7 msec are needed, because of the time required to write the modified object to stable storage. Actions that make handler calls to other guardians take considerably longer to run. A topaction that makes one null handler call before committing needs 36.5 msec to execute. If the handler modifies an object at the remote guardian. then 82 msec are required. Additional measurements of the performance of Argus can be found in [7].

7.3 Cost of Orphan Detection — Individual Operations

There are several costs associated with orphan detection that need to be considered. First, there is the cost of transmitting orphan detection information in messages. In our algorithm, this includes the cost of piggybacking *done* and the timestamps for *map* and *gmap*. Because timestamps are small, the main concern is the cost of transmitting a large *done*. Next, there is the cost of processing *done* when a message is received. We must also consider the cost of communicating with the central server, and of processing the server's

replies.

7.3.1 Cost of Piggybacking Done

In Figure 7.1, the cost of including *done* in messages is given. The figure shows the round trip time for a low level send and reply when *done* information is piggybacked on the send message. In this example, the reply is a null message not containing *done* information. The time given includes the time needed to construct and transmit each message, and the time needed for the recipient to read it, but not the time needed to process the *done* information. The values given in the figure do not include garbage collection time; we did not allow garbage collection to take place during a measurement, because its effect is too variable. The figure shows the relationship between message round trip time, size of *done* and the size of each *aid*. In this set of tests, all *aids* in *done* at a given time had the same length; a different *aid* size was used in each test. The shortest *aids* used were those representing non-nested topactions; these were 16 bytes in length. The longest *aids* represented deeply nested actions with nine ancestors, and were 88 bytes long.

Each line in Figure 7.1 is a best fit line, obtained from 11 points, representing *done* sizes of 0, 10, 20, ..., 100. These lines were calculated using the method of least squares. For each *aid* size, the correlation between *done* size and message round trip time was greater than 0.99. The slopes of these lines are 280μsec/entry when *done* contains only topaction ids, 370μsec/entry for *aids* with two ancestors, 480μsec/entry for ids with 5 ancestors and 650μsec/entry for ids with 9 ancestors.

As illustrated in Figure 7.2, the cost of piggybacking *done* is essentially the same for the central server and deadlining optimizations. This is to be expected, because the structure of *done* and the operations involved in transmitting and receiving messages are the same in both versions of the algorithm. In each case, the round trip time may be approximated by the following formula:

$$\text{time} = 15\text{msec} + \text{size-of-done} * (240\mu\text{sec} + \text{aid-size} * 40\mu\text{sec})$$

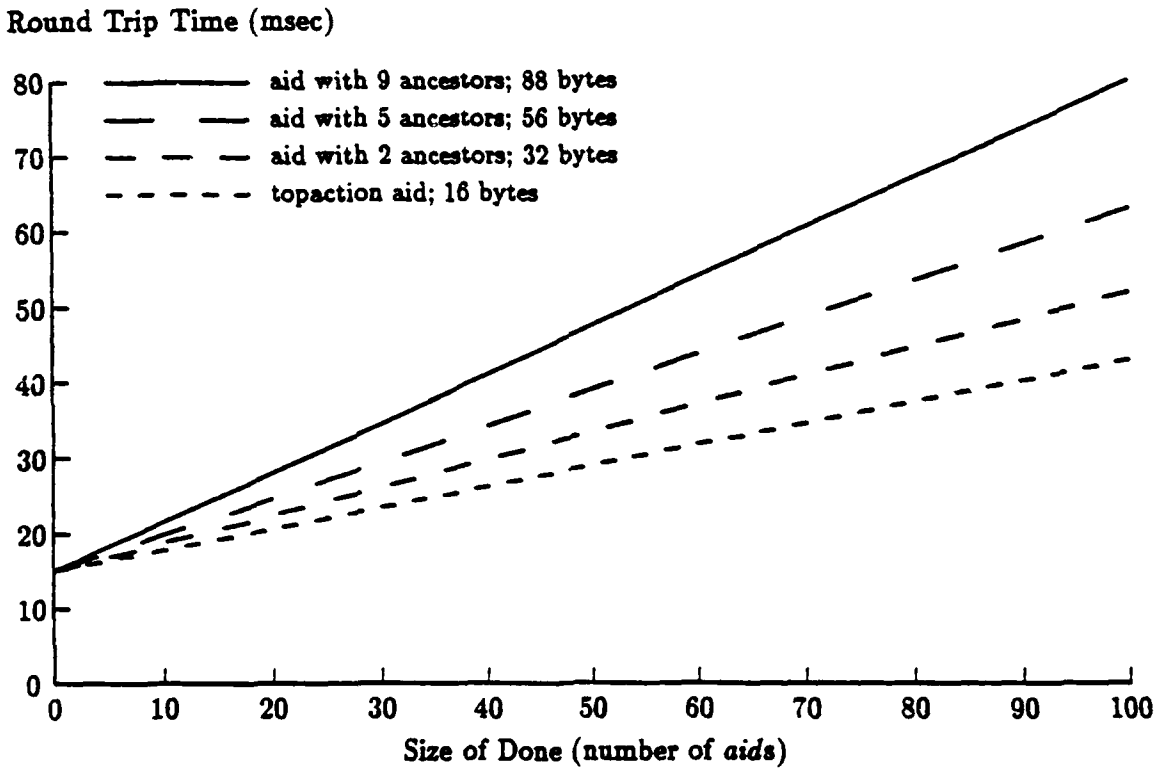


Figure 7.1: Cost of Piggybacking Done

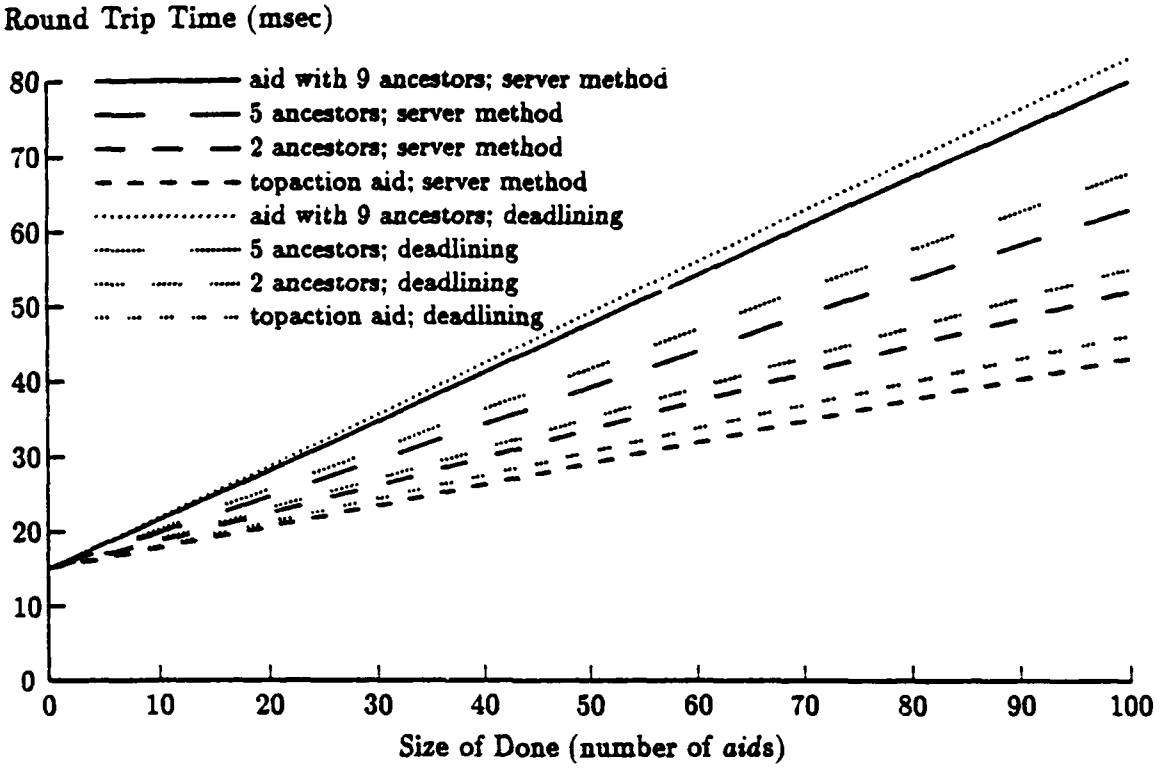


Figure 7.2: Cost of Piggybacking Done: central server algorithm vs. deadlining

In the formula, size-of-done is the number of entries *done* contains, and aid-size is the number of components (⟨gid, sequence number⟩ pairs) that each *aid* has. The formula was obtained by performing linear regression on the slopes of the regression lines for different *aid* sizes. The slope of this second regression gives the 40μsec figure in the formula; the intercept gives the 240μsec figure. The 15 msec value is the mean intercept of the original regression lines.

The formula reflects the fixed cost of transmitting a message, the fixed cost associated with each *aid* (e.g. cost of procedure calls to the routines to read and write each *aid*), and the time needed to read and write each *aid* component. Note that *done* contains the same amount of information in both versions of the algorithm. In our version, each *aid* is four bytes longer than in the deadlining implementation, because the generation number is included. In the deadlining scheme, however, each *done* entry includes a four byte deadline

in addition to the *aid*; therefore, *done* entries are the same size in both cases.

7.3.2 Cost of Merging Done

Next, we examine the cost of merging a copy of *done* received in a message with the version a guardian already has. In our experiments, the cost of merging two "similar" copies of *done* is considered. We consider two versions of *done* to be "similar" if they contain the same set of entries. (In Argus, two distinct objects having the same value are said to be "similar" but not "equal".) Tests were run using similar versions of *done* because this is expected to be the most common case. Because *done* is included in nearly every message between guardians, the guardians' copies of *done* should not differ very much, if they communicate frequently.

This time needed to merge two copies of *done* depends on several factors. The most obvious of these is the number of entries that *done* contains. The size of each *aid* in *done* also matters, because merging involves many comparisons of *aids*, and larger *aids* take longer to compare. A third factor is the time needed to perform lookups in the generation map. Before an *aid* is added to *done*, its generation number is compared with the *gmap* value of its origin. If the *aid* has a smaller generation number, then it is not added to *done*, because its descendants can be detected as orphans using *gmap*. Because we have implemented the generation map as a sorted array, the time needed to perform a *gmap* lookup, using binary search, should be logarithmic in the size of *gmap*. Other implementations (e.g. hash table) would allow faster lookups for large *gmaps*, but might increase the space required, or the time needed for other *gmap* operations. In our experiments, we determined the time needed to merge two copies of *done* both for a small generation map (only one entry) and a large one (one hundred entries).

An additional cost in merging *done* is the time needed to determine whether orphans are present in the system, and to destroy them. In the case of similar *dones*, no actions can be detected as orphans, so this factor does not affect the merge time. In the more general case,

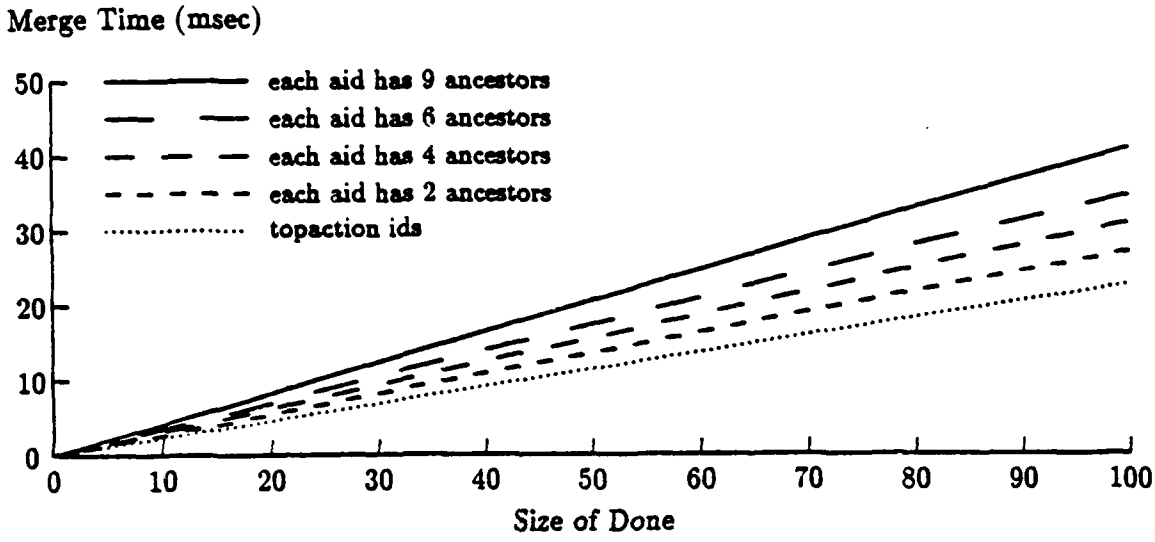


Figure 7.3: Cost of merging two similar dones when gmap contains only 1 element

it is difficult to determine the time needed to find and kill orphans, because this depends on the number of active threads within the guardian that have to be examined and on the time each orphan runs before destroying itself.

Figure 7.3 gives the time needed to merge two similar copies of *done* when *gmap* contains only one entry. The figure consists of the regression lines for merge time vs. *done* size, obtained using the method of least squares. Figure 7.4 gives the merge times when *gmap* has one hundred entries. The correlation coefficients obtained were almost exactly equal to 1 (0.997 or larger).

The cost of merging *done* when the deadlining optimization is used is given in Figure 7.5. From this figure, it is clear that merging *done* takes longer when the central server optimization is used. The main reason for this is the cost of looking information up in the generation map. This cost is significant even when *gmap* is small. To minimize the time required, the procedure call to the *gen_map* data type's *lookup* routine was inlined. However, time is still needed to fetch the origin and generation number from each *aid* in the incoming *done*, search the generation map for the origin's *gid* (using binary search) and compare the origin's generation count with the *aid*'s generation number. These few

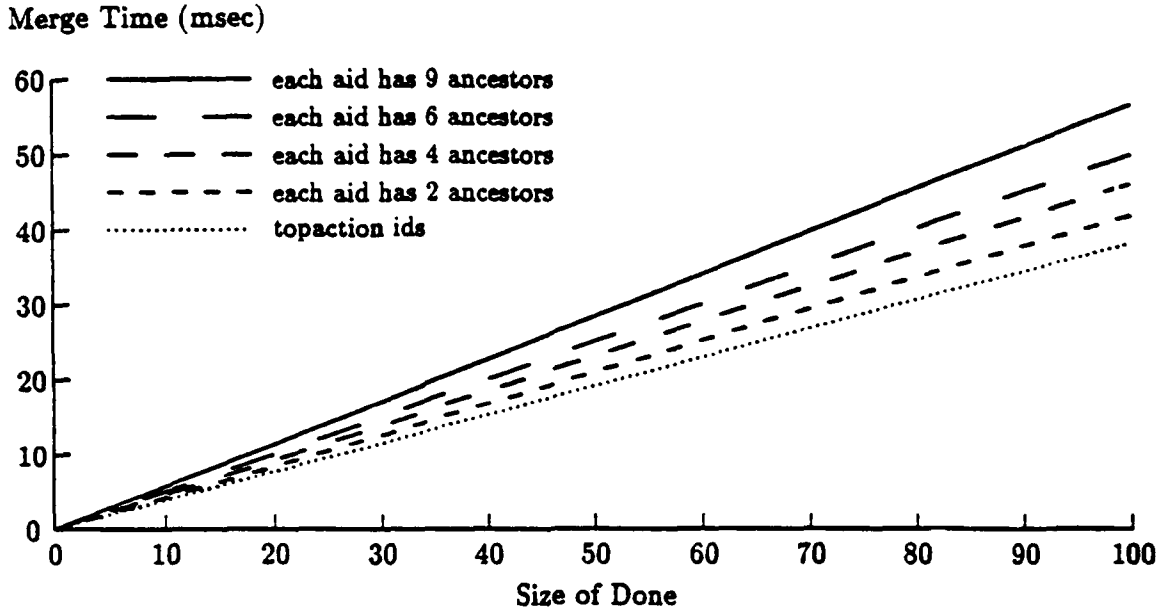


Figure 7.4: Cost of merging two similar dones when gmap has 100 elements

extra integer comparisons per entry are enough to subtantially increase the merge time. Figure 7.6 illustrates the difference in merge time for the two algorithms. This figure shows the time needed to merge *done* for the deadlining algorithm and the central server scheme; it also shows the merge time for the server method when the *gmap* lookup is omitted.

7.3.3 Communication With the Map Service

The next orphan detection cost we will examine is communication with the central server. Figure 7.7 shows the time needed to send a map request to a server and receive a response. This includes the time to construct and transmit the request, processing time at the server, and the time needed to receive and read the reply. The figure shows that the *insert* operation take much longer than *refresh*; the reason is the stable storage write that *insert* requires.

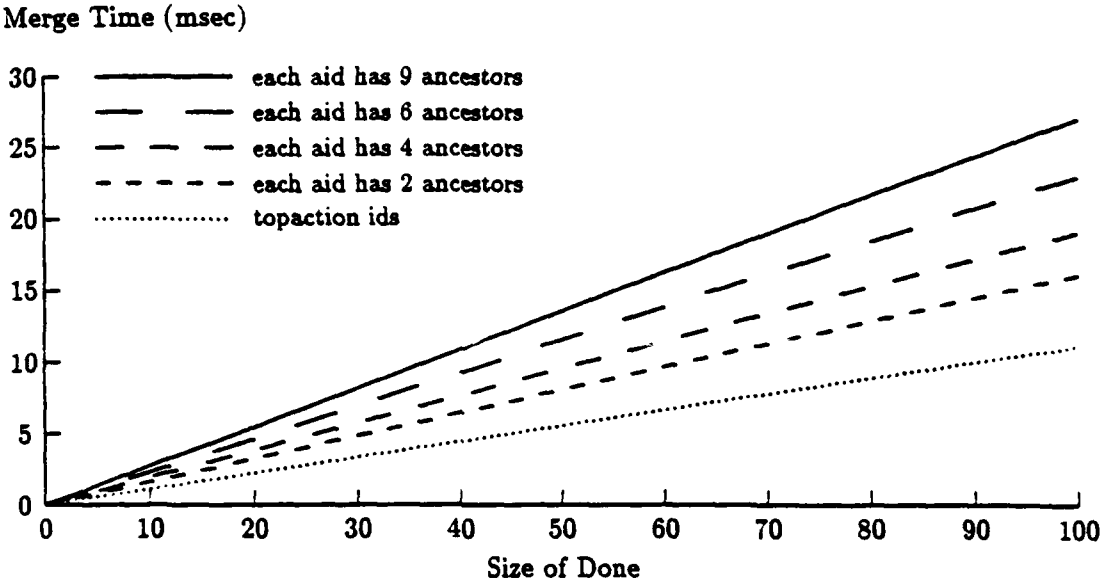


Figure 7.5: Cost of merging two similar dones for the deadlining algorithm

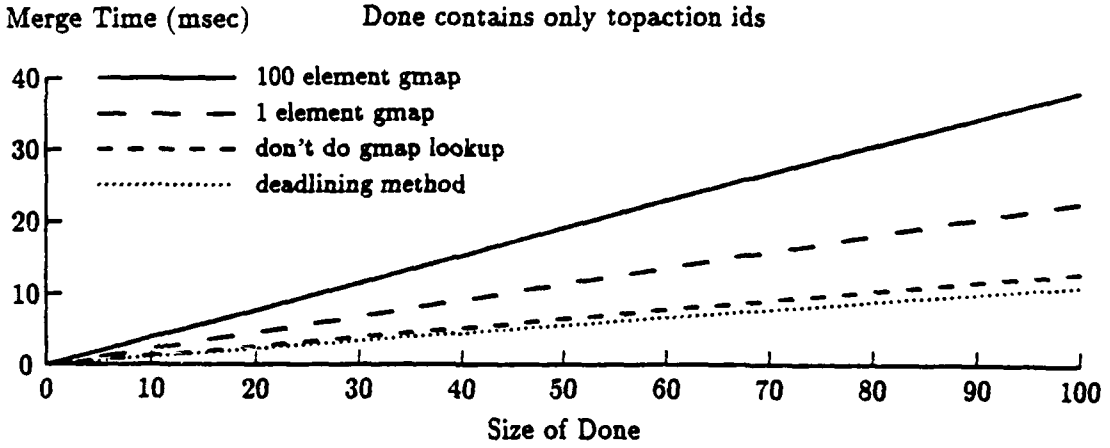


Figure 7.6: Cost of merging done: server method vs. deadlining

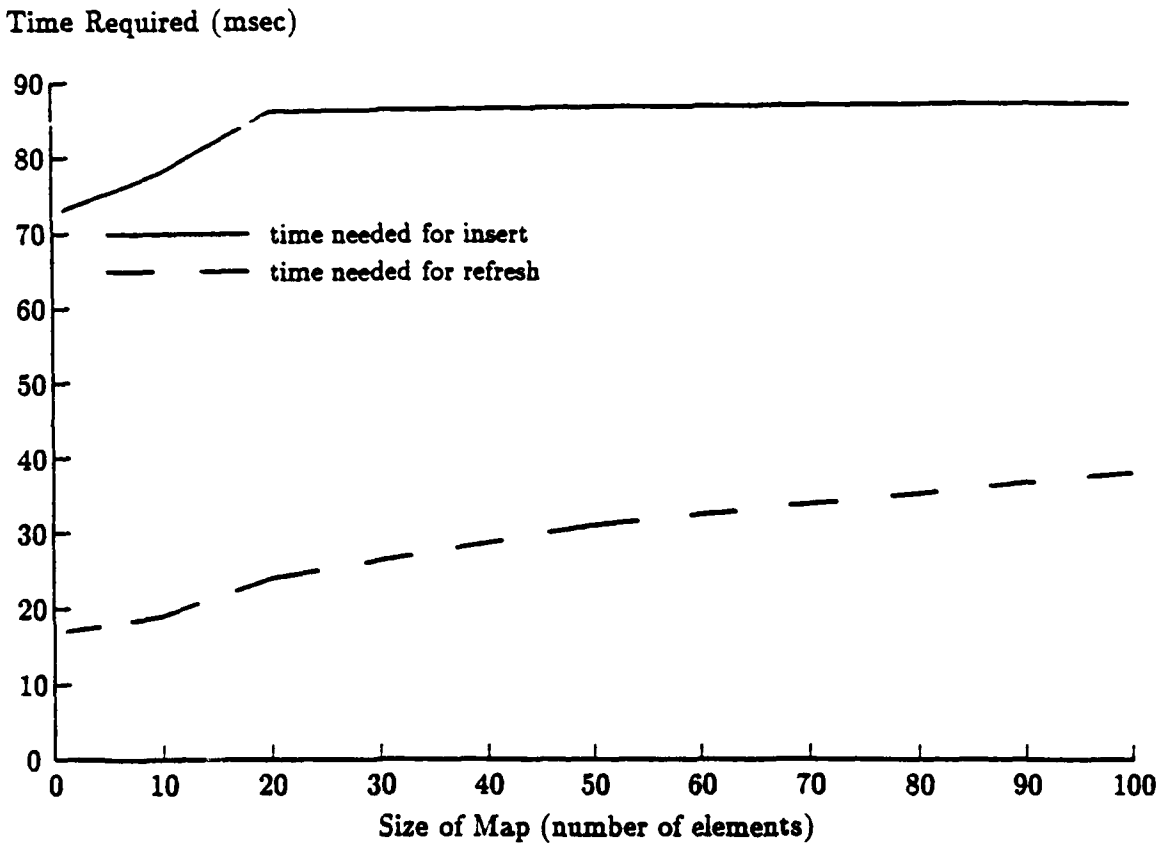


Figure 7.7: Time required for server operations

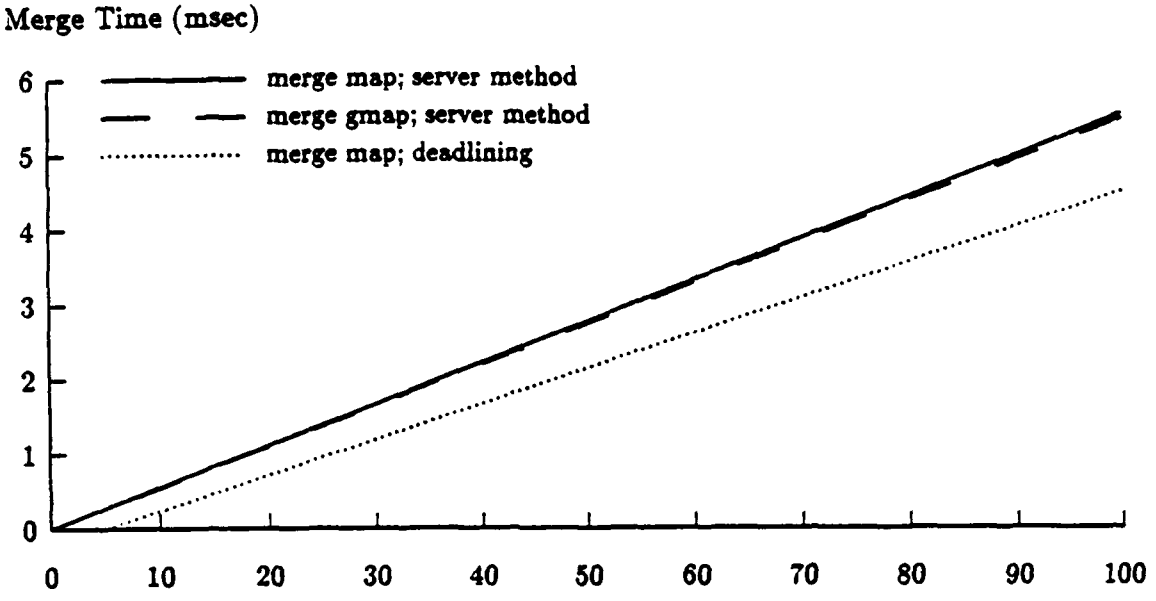


Figure 7.8: Time to merge two similar maps or gmaps

7.3.4 Processing Map and Gmap

Another cost of the orphan detection algorithm is the time needed to merge a copy of the map or generation map received from the central server with the version the guardian already has. Figure 7.8 indicates the time needed to merge two maps or generation maps. Garbage collection time is not included; garbage collection was not allowed during a measurement. In each case "similar" *maps* or *gmaps* were used. This is realistic for *map*. If guardian crashes are relatively rare, few entries should change between successive calls to the map service, and the old and new maps should be "almost similar". For the generation map, the number of differences between the old and new maps depends on the frequency of generation updates and refreshes. However, if guardians are not created and destroyed very often, the generation maps should differ mainly in their generation values, not in the gids they contain, and the merge time should be essentially the same. The additional cost in this case is the time needed to find and destroy orphans dependent on the guardians whose generation counts have changed. As mentioned earlier, this cost is difficult to measure.

Merging maps involves essentially the same operations as merging generation maps, so we would expect the time required to be nearly the same. The figure shows that this is true. The cost of combining two maps (or *gmaps*) is linear in their size. This is as expected, because the sorted array representation of *map* and *gmap* allows a linear time merge.

In the figure, we also give a regression line for the time needed to merge maps when the deadlining optimization is used. As explained in [12], the negative Y-intercept of the regression line is a result of random variations in the data. Clearly, the regression line does not accurately describe the time needed to merge small maps. For large maps, the merge time for our algorithm is slightly larger than for the deadlining scheme.

Even though merging maps takes slightly longer when the central server scheme is used, the cost resulting from merging maps is less than for the deadlining method, because maps are merged much less frequently. When our method is used, maps are merged when new map information is received from the map service. If crashes are rare, this should not happen very often. In the deadlining scheme, the map is piggybacked on almost every message between guardians, creating a larger total cost.

An additional cost related to orphan detection is the time needed to remove old entries from *done* when a new generation map is received. The time required depends on the number of elements in *done* and the time needed to look up their generations in *gmap*. Figure 7.9 gives the relationship between size of *done* and deletion time, for large and small *gmaps*. The values given in the figure are for the case when all *done* entries are for topactions and no entries actually need to be removed. (The time needed to remove old entries is independent of the size of the *aid*s and the number deleted.) We delete old elements by scanning the *done* array and comparing each *aid*'s generation number with its origin's generation count. If the ith *aid* does not need to be deleted, a pointer to it is stored in the $(i-k)$th position in the *done* array, where k is the number of elements deleted so far. If the *aid* is old, we simply go on to the next id. Finally, the array is trimmed to the size of the new *done*.

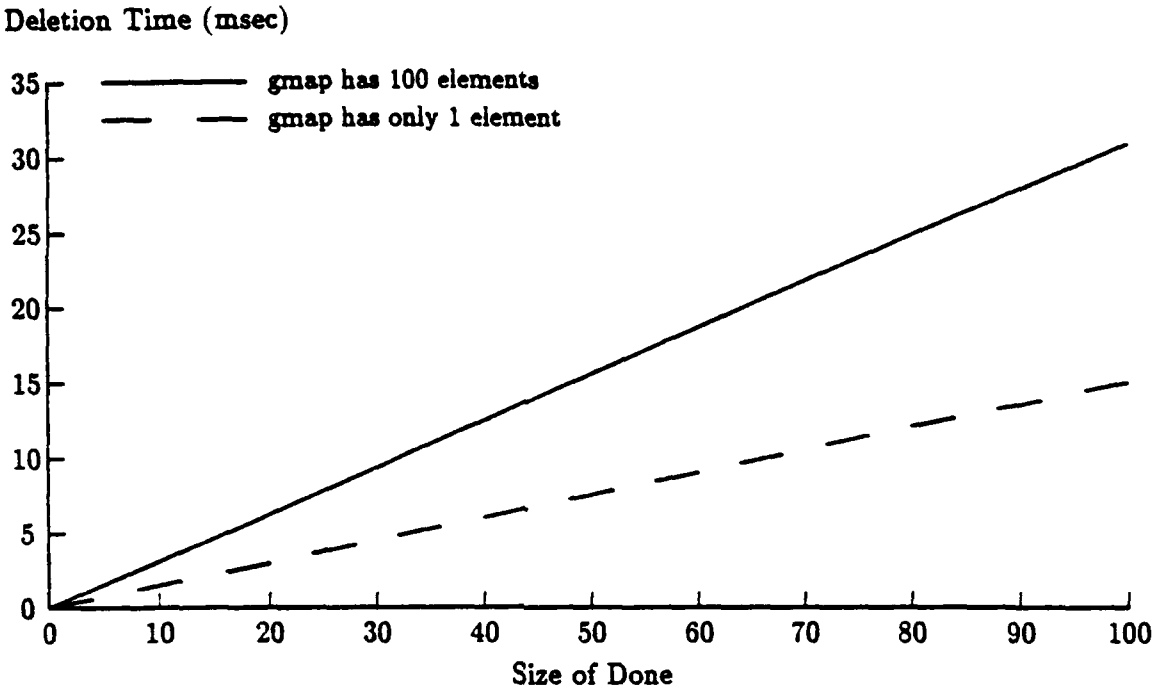


Figure 7.9: Time needed to delete old entries from done

7.4 System Performance Under Selected Loads

In this section we examine the actual performance of the Argus orphan detection algorithm under different loads. Our tests of the system's performance deal entirely with the abort orphan part of the algorithm; the effects of crashes and *map* information are ignored. There are two reasons for concentrating on abort orphans. One is the greater importance of abort orphans in determining system performance. Guardian crashes should be relatively rare, so *map* information should change much less rapidly than *done* and *gmap*. Therefore, guardians should communicate with the generation service considerably more often than with the map service, and abort orphans should be more numerous than crash orphans. For this reason, the performance of the system should be determined mainly by the behavior of the abort orphan part of the algorithm.

Another reason for measuring the performance of only the abort orphan part of the algorithm is the limited number of guardians that could be created for these tests. Only

a limited number of machines were available for the tests, and only a small number of guardians could run on a single machine simultaneously. When too many guardians are created on one machine, interference between them can seriously affect performance. The results of such tests may not be applicable to the case of only one (or a few) guardians per node. However, with only a small number of guardians in each test, the number of crashes occurring during an experiment would be too small to be interesting, unless an unrealistically high crash rate was used.

The tests described in this section were run using a modified version of a load generator developed by Thu Nguyen [12]. This program was originally developed to aid in measuring the performance of the deadlining optimization of the orphan detection algorithm. The load generator consists of three components: a load manager, to interact with the user and run specified tests, a load compiler, which accepts a job specification indicating the number and types of guardians to be created, and a generic guardian type. The load manager uses the compiler to compile a test description, creates a generic guardian for each guardian in the test, and passes each guardian the compiled description of its behavior. The guardians are then allowed to run for a specified period of time. Each guardian carries out its specified behavior, and collects data about events related to orphan detection. The behavior of each guardian is defined in terms of events relevant to orphan detection: frequency of action creation, abort rate, frequency of handler calls to each other guardian, time spent busywaiting, etc. This information is provided for the background thread at each guardian, and for each handler the guardian has. In addition to carrying out its specified behavior, each guardian collects data concerning the performance of orphan detection; this data is sent to the load manager before the guardian is terminated.

A full description of the load generator will not be given here; we will only indicate the general nature of our modifications to it. The main changes involved altering the generic guardian type to use our orphan detection modules, and to record the data appropriate for the central server algorithm instead of for the deadlining scheme. For example, information about frequency of communication with the central server was added; measurements related

to deadlines were omitted as no longer applicable.

7.4.1 General Characteristics of System Tests

In most of our system tests, the pattern of communication among guardians involves one or more client guardians making calls to a single server. This is a realistic pattern for many applications. We also consider cases of related subsystems, each consisting of a server and clients. In these tests, each client communicates only with the server for its subsystem; servers for different subsystems occasionally communicate with each other. It is important not to confuse the "server" guardians in our experiments with replicas of the map and generation services. Both the "client" and "server" guardians of our experiments are clients with respect to these services. To avoid confusion, in this discussion, the terms "client" and "server" will refer only to the ordinary guardians in our experiments. The term "replica" will be used to refer to map service and generation service replicas.

In each of our experiments, a single replica was used for the map and generation services. Increasing the number of replicas should not affect performance very much. If several replicas were used, gossip messages could be sent after each update. In this case, replicas would have the same information most of the time. When a client needed a new *map* or *gmap* it would normally be able to obtain it from the first replica it tried. As a result, the performance of a multiple replica system should not be very different from that of a single replica implementation. Although tests could have been run using several replicas, with less frequent gossip or artificially high loss rates for gossip messages, this did not seem worthwhile.

Our tests also do not make use of background queries from the *map_service* to the replicas. If handler calls are made frequently, as in our tests, then normal *refresh* operations will have to be performed soon after any change to the *map*, and background queries will generally not be made anyway. In this case, adding frequent background queries creates more work for the guardian, without actually reducing handler call processing delays, and

without reducing the size of *done*.

To avoid the problem of interference among guardians, most of our tests involved only a single guardian per machine. When the number of guardians in an experiment exceeded the number of available machines, the guardians were distributed among machines to balance the load as evenly as possible.

7.4.2 Quantities Measured

The main quantities measured in these experiments were the number of entries in *done* and the frequency of communication with the server. The time needed to perform important actions, including the duration of topactions and the time needed to communicate with the server, were also recorded. However, timing measurements made during these tests were not reliable. Because many processes besides our guardians run on each machine, our guardians could be interrupted during a timing measurement, while some other process ran. The other process could run for a long time (seconds, while most events we are measuring are only milliseconds long), completely invalidating the measurement. Garbage collection at the guardian also affected the measured time values.

When running the tests described in the previous section, it was possible to obtain more accurate timing measurements. This was accomplished by locking out all other threads at the guardian, and by measuring resource usage by the guardian's process (using the appropriate UNIX system call) rather than real time. When real time measurements were needed, in order to measure the time required for communication with the server, many short trials were run, with garbage collection disabled during a trial, and an average value for all trials was then calculated.

It should be noted that all tests were run on time sharing systems, where the presence of other users may affect results either by interrupting events and affecting the time needed for them to complete, or by slowing down the guardians. If a guardian runs more slowly

than expected, it will create fewer actions per unit time than expected, and will have fewer *done* entries than otherwise. To minimize this interference, tests were run when other users were not on the machines involved. However, some interference must still be expected, from system processes and from network traffic between machines not involved in the experiments.

7.4.3 Issues in Performance Measurement

Several problems had to be addressed in order to obtain meaningful results from our experiments. One problem is that the size of *done* is not constant throughout an experiment. *Done* continually grows as actions abort, and then shrinks suddenly when a guardian advances its generation count in *gmap*, allowing its old *done* entries to be purged. To obtain accurate estimates of the long term average size of *done*, without running very long experiments, it was necessary to set the running time of each test equal to a multiple of the generation update interval.

Another problem to be considered is that generation advancement by different guardians may become synchronized. If each guardian uses the same generation update interval, then their generation updates can stay synchronized throughout an experiment. In this case, if one guardian advances its generation shortly before a second guardian does, the second guardian may perform its update before information has propagated to it from the first guardian via handler calls. Thus, the second guardian learns about the change to the first guardian's generation count when it receives the result of its own update operation. The first guardian, on the other hand, needs to use the *refresh* operation to learn about the second guardian's generation change. Later, the first guardian must call the generation service a second time its generation count is increased again. Thus, two guardians in a system that at first appears symmetrical have very different behavior. One guardian needs to talk to the generation service twice as often as the other. On the other hand, if the two guardians did not perform their updates so close together, each would need to talk to

the server with the greater frequency. Thus, very different performance may be obtained depending on precisely when the two guardians perform their updates.

Clearly it is undesireable for the results of our experiments to depend heavily on circumstances such as the precise timing of the updates, which may vary each time an experiment is run. In a real system, the guardians probably would not be perfectly synchronized, and would not constantly call the service at nearly the same time. To obtain more realistic results, we adopted the following procedure. In experiments in which there were exactly two guardians that performed generation updates, we noted whether one guardian never (or almost never) performed refreshes. If so, we inferred that this guardian was performing updates immediately after the other guardian. In this case the experiment was run again, to get data for the normal case. We observed that in either case, the size of *done* was the same. In addition, in the special case, the guardian that performed its updates first had the same refresh rate as guardians do in the normal case. For this reason, only data for the normal case are presented here.

When there are many guardians, it is more likely that, even in a real system, some updates will occur at almost the same time. In these cases, we simply ran the experiment many times, and took averages over the different trials and the different guardians in each trial.

A third difficulty in measuring performance arises if action aborts occur randomly. We would like to examine the behavior of a system where a certain fraction of handler calls abort and the rest commit. Each call could be modelled as an independent Bernoulli trial. However, this introduces randomness into our measurements of the size of *done*. Clearly, the more actions that happen to abort in a particular test run, the larger *done* will be. Unless very long experiments are run, there will be a lot of variation in the results. To obtain more consistent results without longer experiments, we aborted actions regularly, instead of at random. For example, if a one percent abort rate was used, then every one-hundredth action was aborted. This is not an ideal solutionm, but seems unavoidable unless very long

experiments are run. The same solution was used in Nguyen's research on the deadlining optimization [13].

7.4.4 Test Results

Our first test load consisted of a single client making handler calls to a single server. Calls were made every 500 milliseconds. The service time for each handler call was set to zero (i.e. call does nothing), except that calls selected to abort were given non-zero service times. This ensured that a call did not commit before we had a chance to abort it. A fixed handler call abort rate of one percent was used. In different runs of the test, the interval at which the generation service was notified of generation changes varied from five seconds to four minutes. The specific values used were 5, 10, 20, 30, 40, 60, 90, 120, 180, and 240 seconds. Each test was run for ten minutes, except for the intervals of 90, 180 and 240 seconds; these were run for nine or twelve minutes, so the experiment length would be a multiple of the update period.

Our second test consisted of two clients making handler calls to one server. As in the first test, calls were made every 500 msec and had zero service time; the abort rate was one percent. The values presented are for the case when the two clients do not notify the server of generation changes at almost exactly the same time. A third test was run under similar conditions, but with four clients instead of two.

The results of these tests can be found in figures 7.10 and 7.11. The values shown were obtained by averaging several runs with each generation update interval. The results for any single run varied from the average by up to several percent, because of the different factors mentioned above.

Figure 7.10 indicates the average size of *done*, measured at the server guardian. The figure shows clearly the expected linear relationship between generation update interval *done* size, except for one anomalous point with four clients and a 60 second update interval.

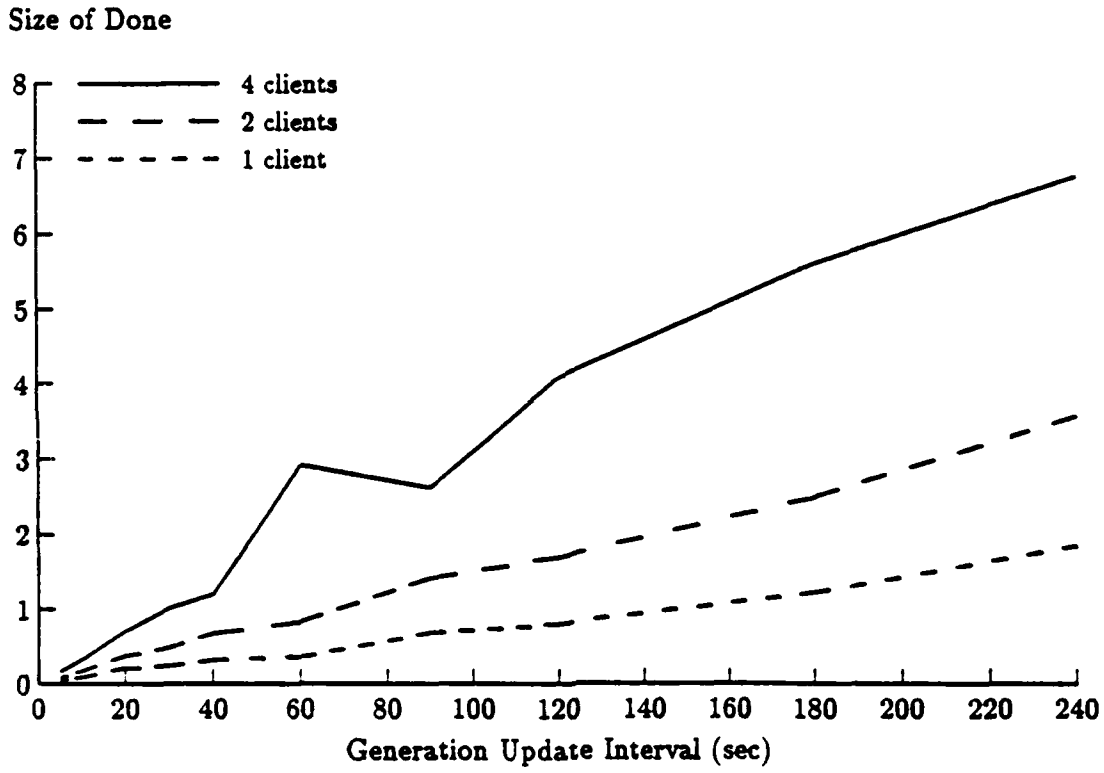


Figure 7.10: Size of done for a client-server system

The size of *done* is also proportional to the number of clients. *Done* remains very small in every case, except for the longest update intervals when there are four clients. To put these values in perspective, recall from the previous section that the round trip time for a message with an empty *done* is approximately 15 msec. Piggybacking a 5 element *done* on the send message increases round trip time by 1.5–3 msec, depending on the size of the *aids* in each *done* entry. Merging two *dones* of this size requires 1–2 msec if *gmap* is small. For a 100 element *gmap*, 2–3 msec are required, but this time could be reduced if a more efficient *gmap* representation was used.

Figure 7.11 gives the probability that a guardian needs to obtain a new *gmap* from the service when a message is received containing a piggybacked *done*. This probability is calculated by dividing the number of *refresh* operations performed by the number of messages containing generation timestamps that the guardian received. In the single client case, the client never performs a refresh operation, because the server is the only other guardian present; the server never runs topactions and its generation count never changes. For this reason, the client's *refresh* probability for the single client system is not included in the figure. The *refresh* probability for the server is essentially constant, regardless of the number of clients. Increasing the number of clients causes the number of refreshes performed by the server and the number of messages processed by it to increase in the same proportion, leaving the refresh probability unchanged. For this reason, the server's refresh probability is given only for a single client system.

As shown in the figure, in the two client case, the refresh probability for the server was approximately half as large as for the clients. In absolute numbers, the server performed twice as many refreshes as each client, because it needed to obtain a new *gmap* after each client advanced its generation. A client, on the other hand, only needed a new *gmap* after the other client advanced its generation count. The server's frequency value is half the value for the clients, because the server processed four times as many messages containing piggybacked timestamps as each client did. The server had to deal with messages from two clients, while each client only received them from one server. The server received both

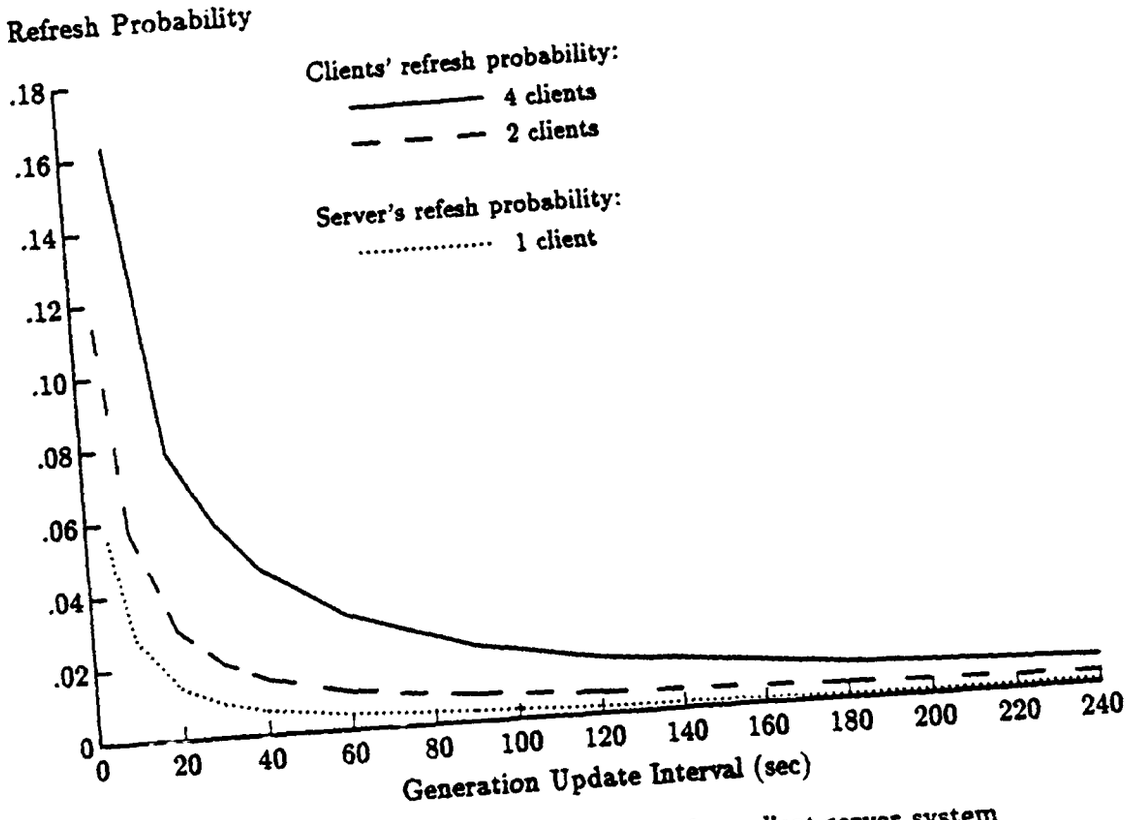


Figure 7.11: Refresh probability for a client-server system

handler call messages and two-phase commit messages containing piggybacked timestamps. The clients only received handler replies with piggybacked stamps; two-phase commit reply messages do not contain piggybacked orphan information. Thus, when computed as a proportion of messages with piggybacked stamps, the server's refresh frequency is half that of the clients. However, when calculated in absolute numbers, or per unit time, it is twice as large.

In the above data, only the *refresh* operations performed by each guardian were considered. The next two figures give the combined frequency of all generation service calls, both updates and refreshes. Figure 7.12 shows the mean time between generation service calls. Because the call frequencies for servers and and clients were essentially the same, only those for servers are given. Figure 7.13 shows the number of generation service calls divided by the number of messages received with piggybacked timestamps. In this way, the cost of updates is amortized over these messages. Because servers process many more messages than clients, their call rate is smaller when expressed as a proportion of messages processed. For this reason, Figure 7.13 includes results for clients and servers separately. As in Figure 7.11, results for the server are given only for the single client case; with different numbers of clients, the server's generation call rate was essentially the same.

In addition to client-server structured tests, we also considered test cases involving partially independent subsystems. Each subsystem consisted of a single server and some clients. Most communication was between clients and the server for the client's own subsystem; occasionally, servers in different subsystems communicate with each other. In these tests, two parameters were varied: the generation update interval and the cross talk rate between subsystems. To keep the number of experiments manageable, we first varied the update interval, while holding the cross talk rate constant; next the cross talk rate was varied for a constant update interval.

The first case of this type involved two subsystems, each with two clients and one server. Each client made a handler call to its server every 500 msec. One percent of all calls were

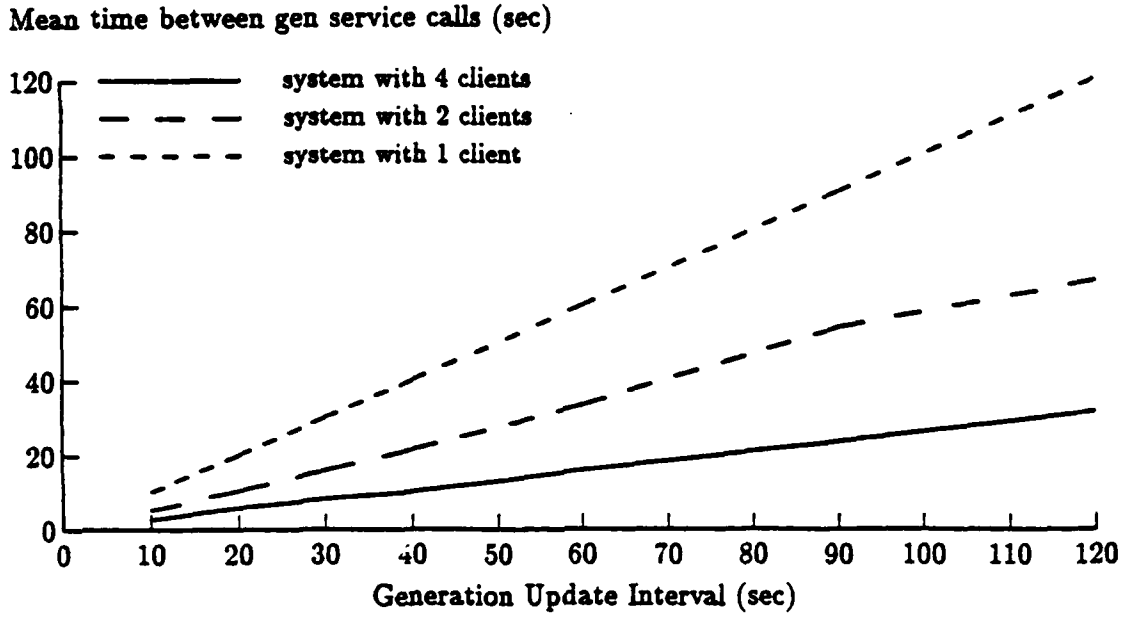


Figure 7.12: Generation service calls for a client-server system

aborted. In addition, the servers communicated with each other once every 50 seconds. Update intervals of 20, 30, 40, 60, 90, and 120 seconds were used. The results for these cases are given in Figure 7.14. Next, the update interval was fixed at 50 seconds, and cross talk intervals of 1, 5, 10, 20, 30, 40, 60, 90 and 120 seconds were used. These results are shown in Figure 7.15. For purposes of comparison, the figures also show the size of done for systems with two or four clients and a single server. Note that for a tightly coupled system, with cross talk more frequent than the per client generation update interval, the size of *done* is close to that of a single server four client system. When cross talk is rare, we approach the single server two client system results. For cross talk intervals comparable to the update interval, we obtain intermediate values.

Note that in this experiment, the servers and clients both create topactions (although the servers' actions do not abort); therefore, they both make *insert* calls to the generation map service. This increases the number of refreshes that all guardians have to perform. With the cross talk interval fixed at a large value, this effect is unimportant. When the generation update interval is long, the servers make very few updates, and provoke few

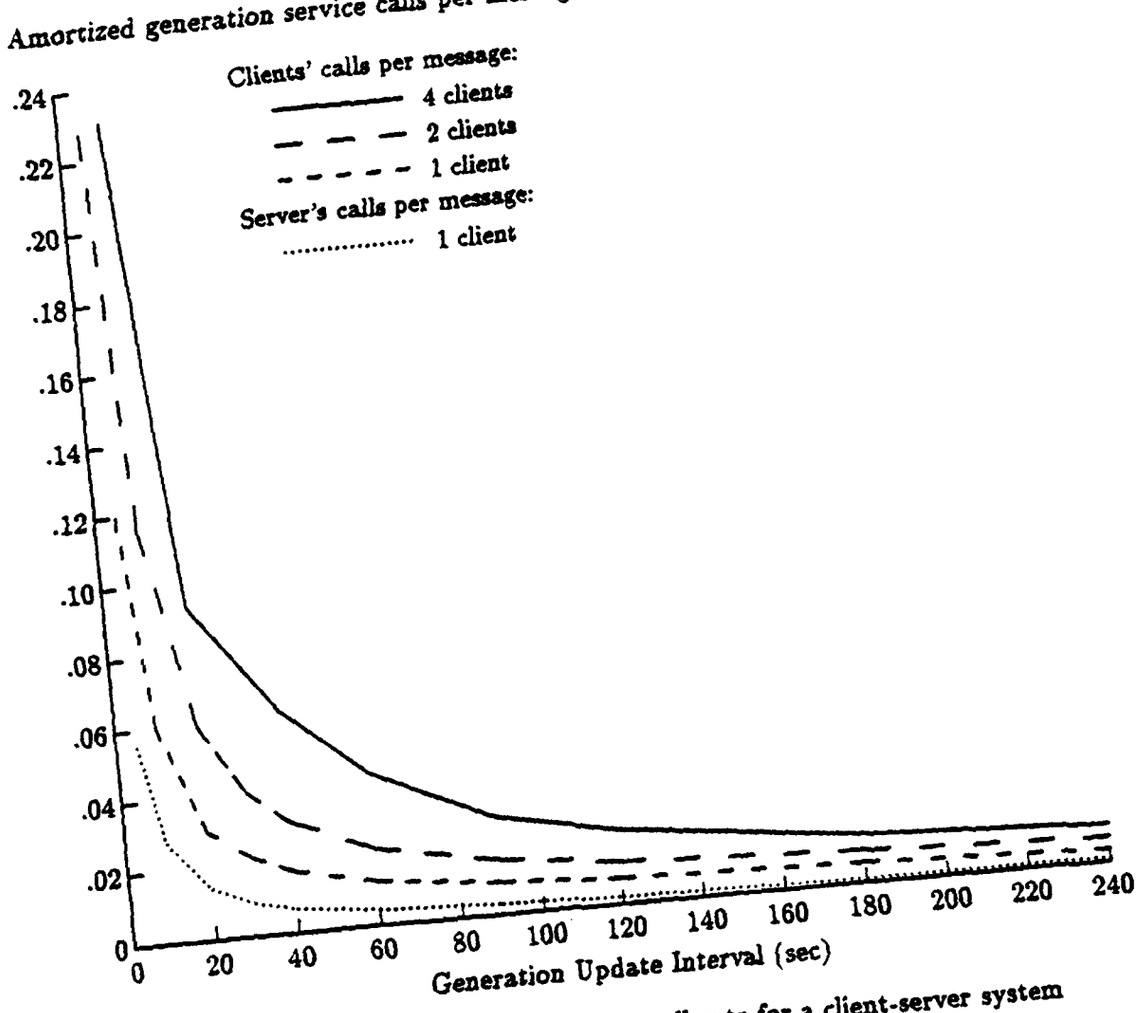


Figure 7.13: Generation service call rate for a client-server system

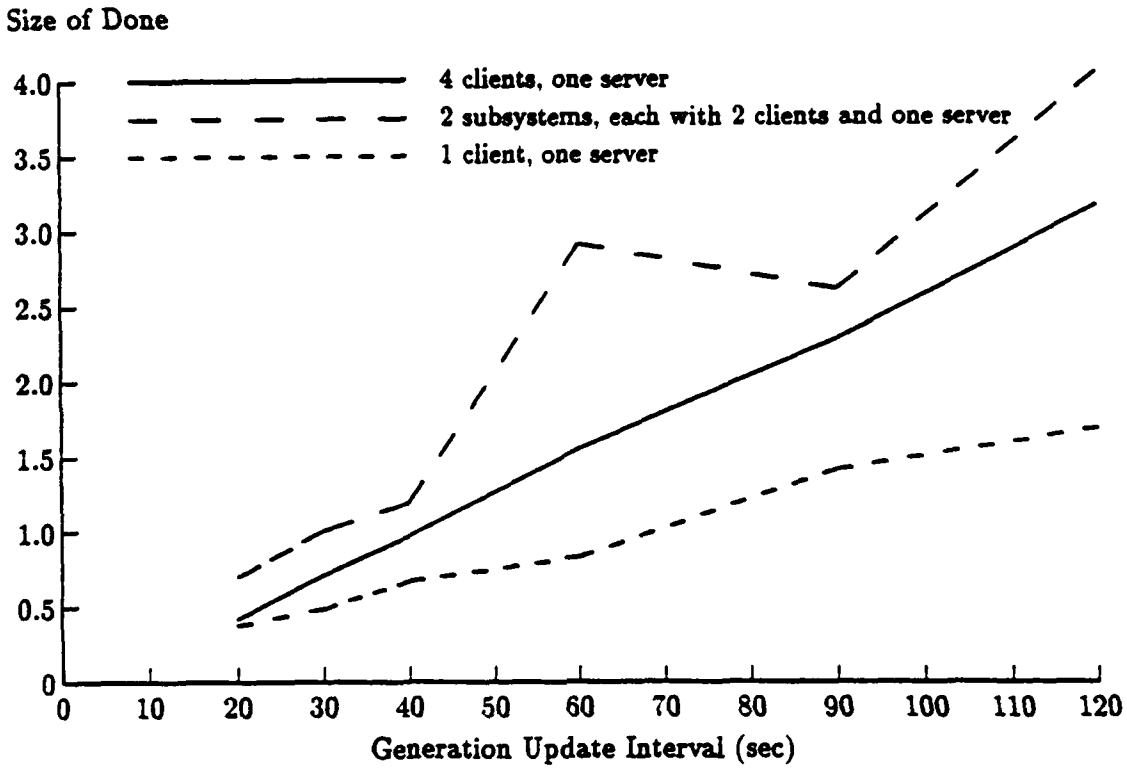


Figure 7.14: Two semi-independent subsystems — cross talk every 50 sec

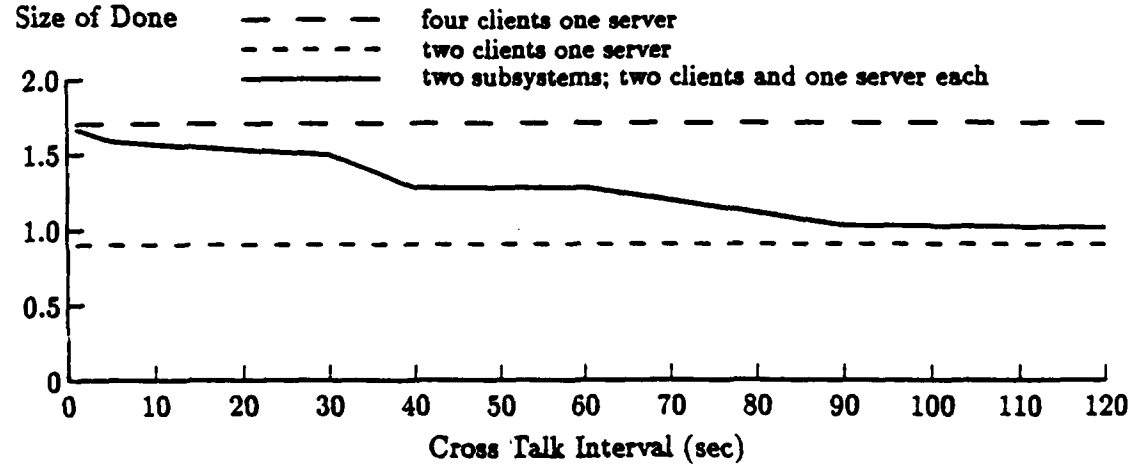


Figure 7.15: Two semi-independent subsystems — update generation every 50 sec

refreshes by clients. When the update interval is short, then most intervals do not include server cross talk, and therefore do not include server updates. When the server cross talk interval is made very small, however, then each update interval includes updates by the servers; thus the total number of updates is increased significantly.

A second case of this type was made using four subsystems, each with four clients and a server. Every server communicated with each of the other servers once during each cross talk interval. Because of the limited number of machines available, it was not possible to use a separate node for each guardian. In order to make the system as symmetrical as possible, and to balance the load, each server was given its own machine. Four other machines were used, each containing one client from each of the four subsystems. The update intervals and cross talk rates used in this experiment were the same as in the previous one. The results are presented in Figures 7.16 and Figures 7.17. The general relationships in this case are similar to the previous one except that the size of *done* is much larger.

Figures 7.18 and 7.19 show the relationships among refresh probability, generation update interval and cross-talk, for two subsystems with two clients and one server each, and for the case of four subsystems, each with four clients and one server. The figures show the refresh probability for a client guardian, for different update intervals and cross talk intervals. As before, refresh probability is the number of refresh operations performed, divided by the number of messages received that contain a piggybacked *done*.

Finally, we consider the effect of different abort rates. Although the rates we consider in this test are unrealistically high, it is still interesting to examine the behavior of the system under these conditions. In this experiment, we test a load consisting of two subsystems, each with two clients and one server. To ensure that cross talk happens at all phases during the growth and shrinkage of *done*, we set the cross talk interval to 50 seconds and the update interval to 60 seconds. Abort rates of 1, 2, 4, 6, 8, and 10 percent were used. These results are presented in Figure 7.20. As expected, the size of *done* grows linearly with the abort rate; the refresh probability is unaffected.

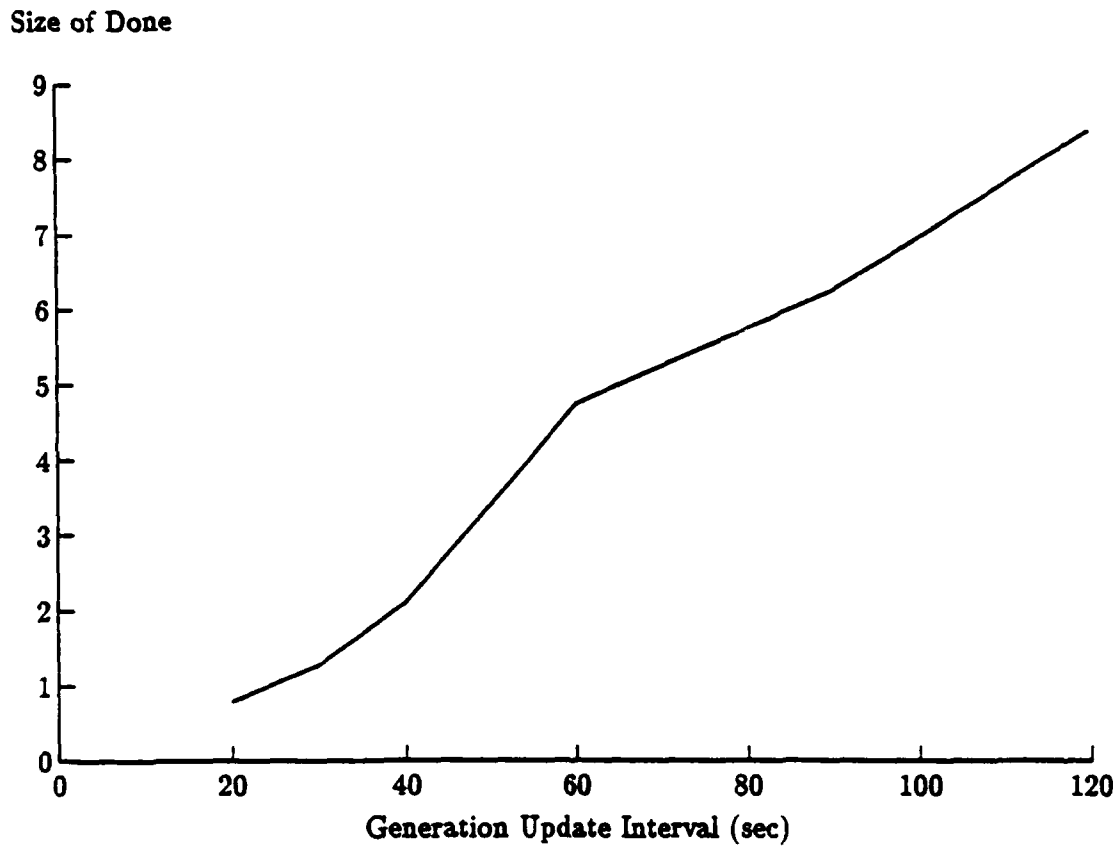


Figure 7.16: Four semi-independent subsystems — cross talk every 50 sec

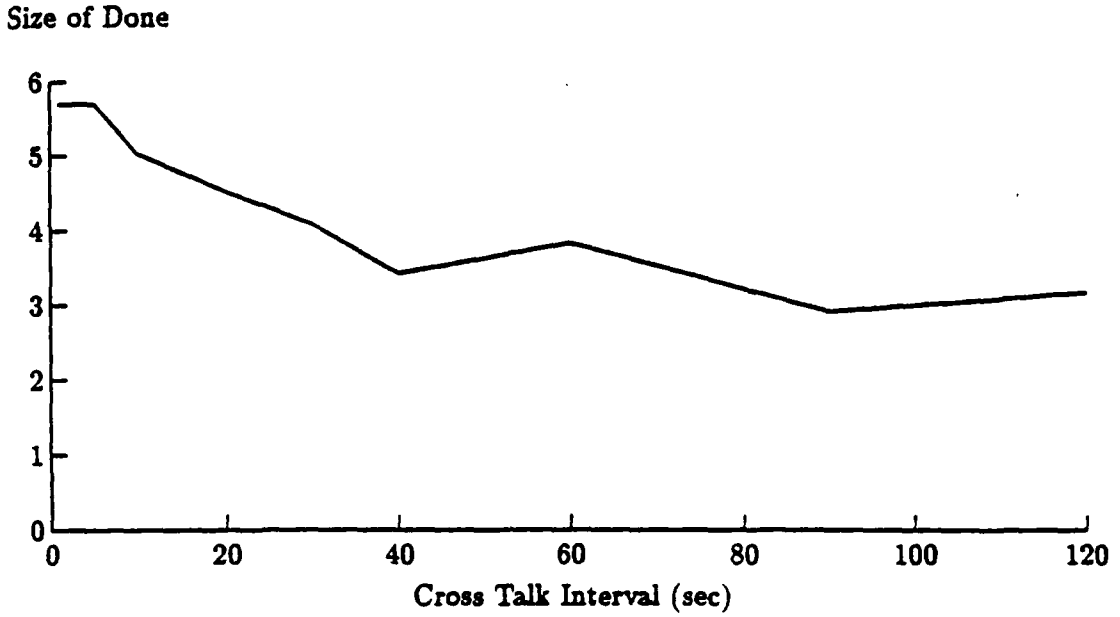


Figure 7.17: Four semi-independent subsystems — update generation every 50 sec

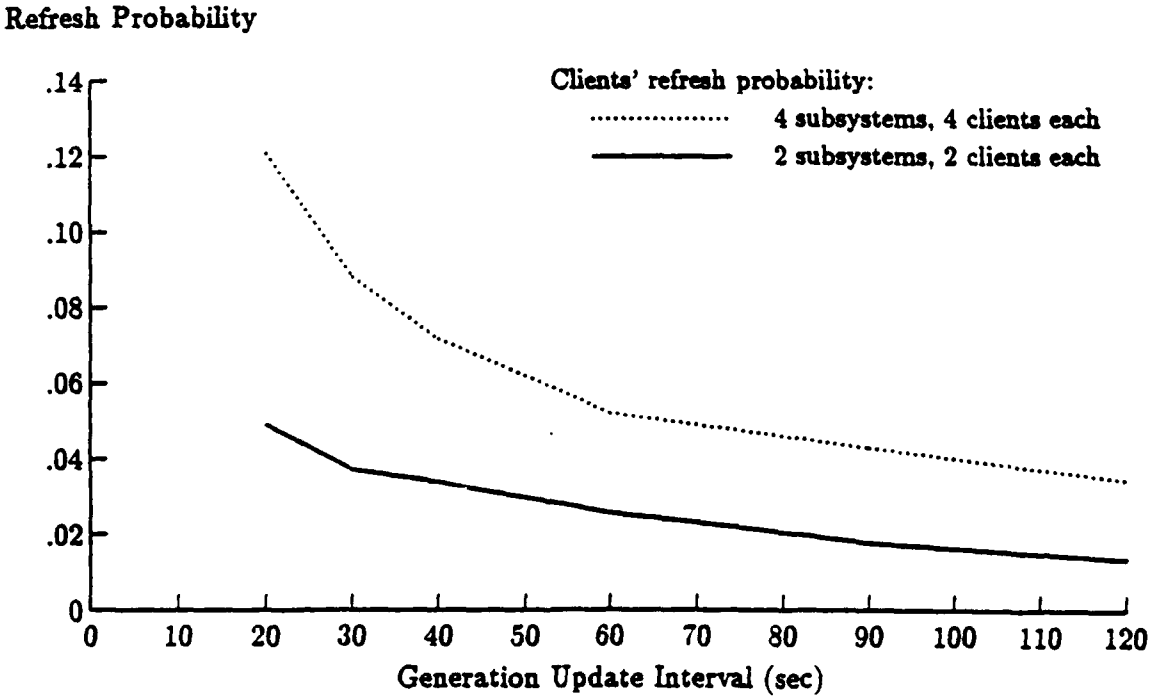


Figure 7.18: Refresh probability for related subsystems — cross talk every 50 sec

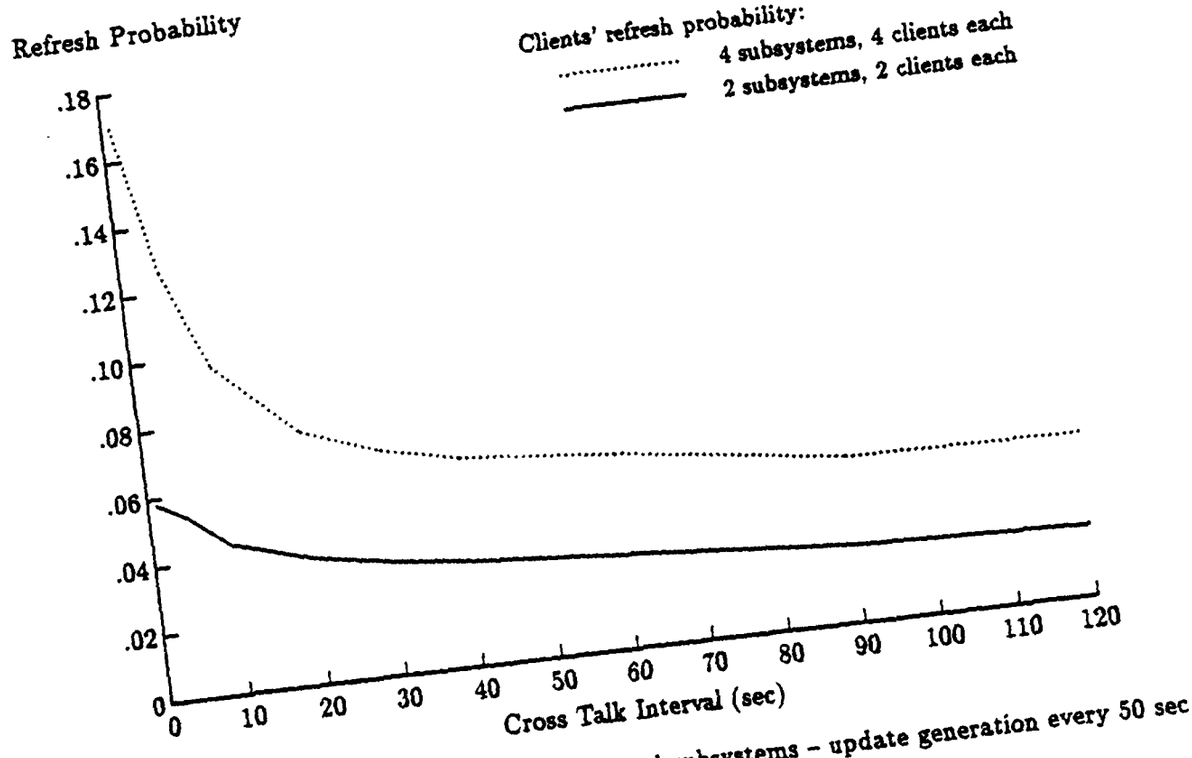


Figure 7.19: Refresh probability for related subsystems – update generation every 50 sec

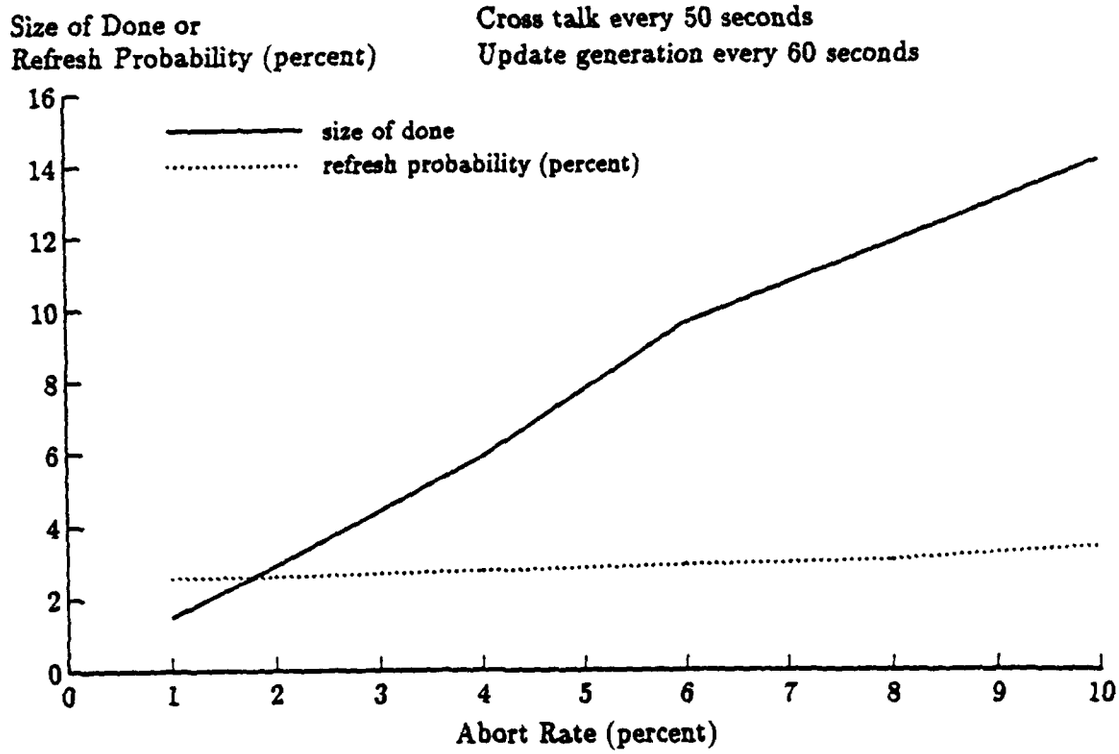


Figure 7.20: Performance for different abort rates — two subsystems

7.5 Comparison With the Deadlining Algorithm

In order to evaluate the performance of the central server optimization of orphan detection, we will compare its performance with that of the deadlining method. The performance results for the deadlining scheme were obtained from [12]. The main issues in evaluating performance are the effectiveness of the two methods in reducing the size of *done* and the amount of additional overhead needed to accomplish this. In the central server method, this overhead results from the need to communicate with replicas of the generation service. In the deadlining scheme, it results from the need to extend the deadlines of actions, to prevent them from terminating prematurely if they run longer than the initial deadline period.

7.5.1 Size of Done

Figure 7.21 shows the relationship between the size of *done* and the *done deadline period*, for a system consisting of one server and one or more clients. The deadline period is the time a topaction is allowed to run before its deadline expires, i.e. when a topaction is created, its deadline is set equal to the current time plus the deadline period. In the test cases described by the figure, each client made a handler call to the server every 500 msec. Except for calls made by actions that were selected to abort, handler calls had a zero service time (or as close as possible to zero). One percent of all handler calls were aborted. For purposes of comparison, the size of *done* using the central server scheme is given, for different generation update intervals. The same scale is used to represent generation update intervals and deadline periods.

The figure shows that when deadline period and generation update interval are equated, the size of *done* is approximately twice as large for the deadlining method as for the central server algorithm. The reason is that with a deadline period of t seconds, *done* contains *aids* for aborted actions from the last t seconds. When the central server algorithm is used, *done* contains *aids* for a guardian's aborted actions from the last 0–t seconds, depending on when the guardian last notified the server of a generation change. On average, *done*

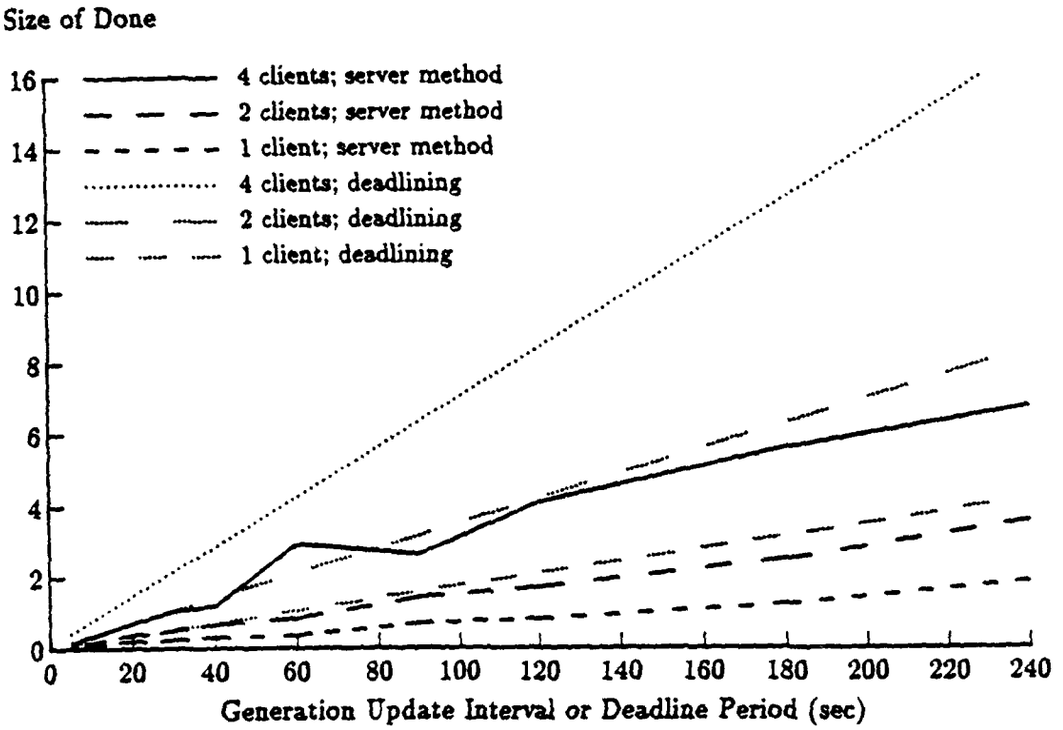


Figure 7.21: Size of done for a client-server system; server method vs. deadlining

contains *aids* for the last $t/2$ seconds, and is half as large as in the deadlining scheme. Note that while the number of *aids* in *done* for a particular guardian grows to a maximum and is then reduced to zero when the generation service is notified of a generation change, *done* as a whole does not grow and shrink as drastically, because different guardians advance their generations at different times.

7.5.2 Total Cost of Orphan Detection

The size of *done* alone is not a sufficient measure of the cost of orphan detection. The two algorithms differ in the amount of overhead they produce, even when the size of *done* is the same. For example, merging two similar copies of *done* takes much longer in our implementation of the server algorithm than in Nguyen's implementation of deadlining. In addition, the central server algorithm involves overhead from communication with the map and generation services. To give a more accurate idea of the performance of the two algorithms we have calculated the total overhead of orphan detection.

The total cost of orphan detection was calculated using the results presented earlier in this chapter. For the deadlining scheme, the overhead included in the calculations is the cost of processing *done*. For the central server algorithm, the cost of processing *gmap* is included as well. The values given are for a system with one server, a small number of clients, and a one percent abort rate for handler calls. Each client starts one topaction every 500 msec, and makes one handler call to the server per topaction.

Only the cost of detecting abort orphans was considered, because we expect this to dominate the cost of crash orphan detection. When deadlining is used, it should be possible to keep the *map* small by choosing an appropriate map deadline period, because crashes are relatively rare. With the central server algorithm, if there are relatively few crashes, guardians should not need to contact the map service very often.

For the deadlining algorithm, the main cost of orphan detection is the time needed to

piggyback *done* in messages, and to merge piggybacked copies of *done* with the versions that guardians already have. For the central server algorithm, the largest costs are piggybacking and merging *done* and communicating with the generation service. The time required for communication with the service was amortized over the total number of handler calls.

The total cost of orphan detection is given by the following formulas. For the deadlining method, the cost per handler call is $27\mu\text{sec}(n)(dp)$, where n is the number of clients in the system, and dp is the *done* deadline period in seconds. For the central server algorithm, the cost is $13.2\mu\text{sec}(n)(ui) + 8.5\text{msec}(n)/(ui)$. In this formula, n again represents the number of clients; ui is the generation update interval. The derivation of the cost formulas can be found in the Appendix.

Figure 7.22 gives the total overhead of the of the two algorithms for different generation update intervals and *done* deadline periods. As shown in the figure, the cost of the server method reaches a minimum at an update interval of 25.4 seconds, regardless of the number of clients. For smaller intervals, overhead is increased because more generation service operations are performed; for larger values, the size of *done* increases, so more time is needed to process it. The minimum overhead is approximately $670\mu\text{sec}(n)$ per call. The cost of the deadlining algorithm is less than this if the deadline period is less than 24.8 seconds.

7.5.3 Extension of Deadlines and Generations

From figure 7.22, it appears that the cost of orphan detection using the deadlining method always decreases as the deadline period gets shorter. This is not really true. When the deadline period is short, actions may need to run longer than their deadlines. In this case, the actions' deadlines must be extended. This does not happen in our example, because each topaction makes only a single handler call, with zero service time. However, if topactions made many calls or if calls ran longer, deadline extension could become necessary. The cost of deadline extension naturally increases as the deadline period becomes shorter,

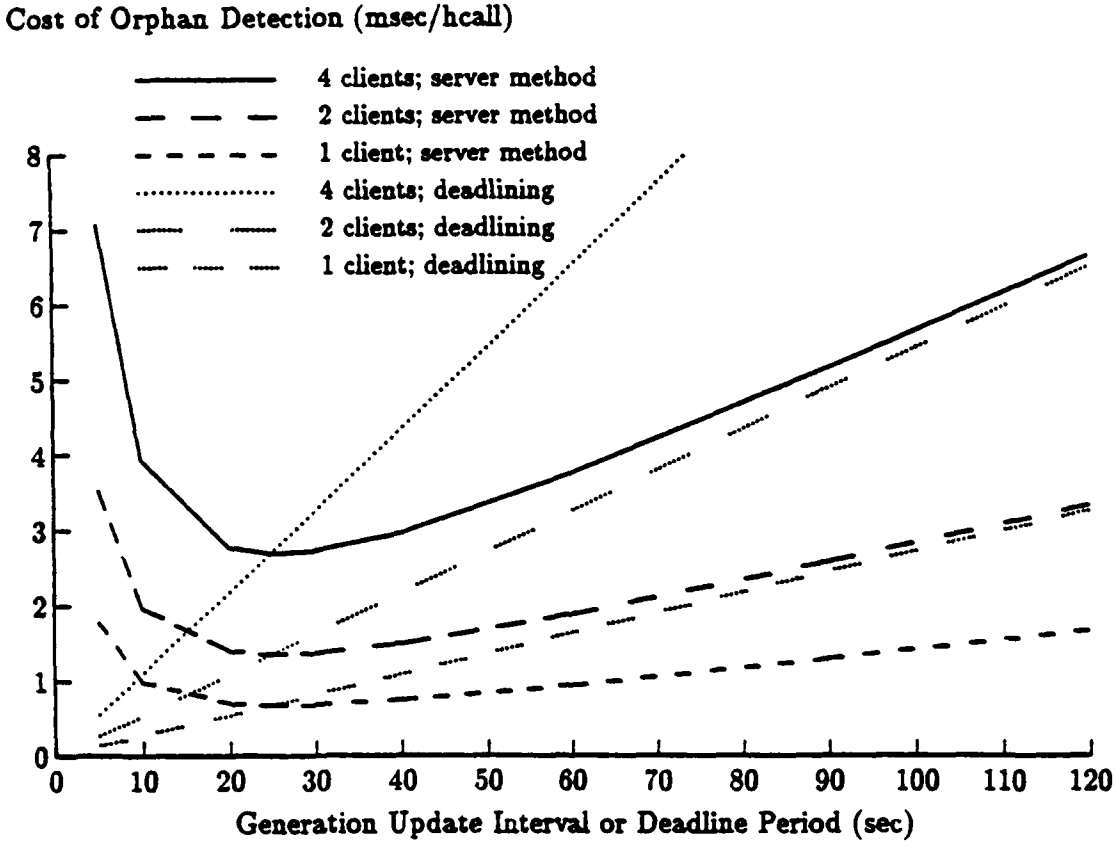


Figure 7.22: Total cost of orphan detection; server method vs. deadlining

because more actions run past their deadlines, and more extensions are required for each action.

The results presented in Section 7.5.2 show that any deadline period of less than approximately 25 seconds produces better performance than the central server algorithm. Therefore, unless actions are very long, it should be possible to choose a deadline period that is long enough to avoid the need for extension, but short enough to give better performance than the central server scheme. Nevertheless, it is interesting to consider the performance of algorithm in the presence of long actions, when deadline extension is required. We also consider the cost of extending actions' generations when the central server algorithm is used.

We do not have data concerning the cost of deadline extension. In [12], information is provided concerning the frequency of deadline extension, but not the time required. In addition, we have not implemented generation extension, and have no data concerning its cost. Therefore, we must estimate the cost of extension.

We will consider a client-server system in which each topaction makes a long series of handler calls. Each call has 500 msec service time. As soon as one call completes, the next call is made. One percent of all calls abort. We consider only the extreme case in which topactions run for a very long time. In fact, we will assume that a single long topaction runs the whole time. Our derivation of the cost estimate for this case is given in the Appendix.

The cost of orphan detection is obtained by adding the extension overhead to the cost in the absence of extension. For the deadlining method, the cost per handler call is

$$18\mu\text{sec}(n)(ui) + xc(ci/dp)$$

For the central server algorithm, the cost is

$$8.8\mu\text{sec}(n)(ui) + (17\text{msec}(n) + xc)/(2ui)$$

The derivation of these formulas is given in the Appendix.

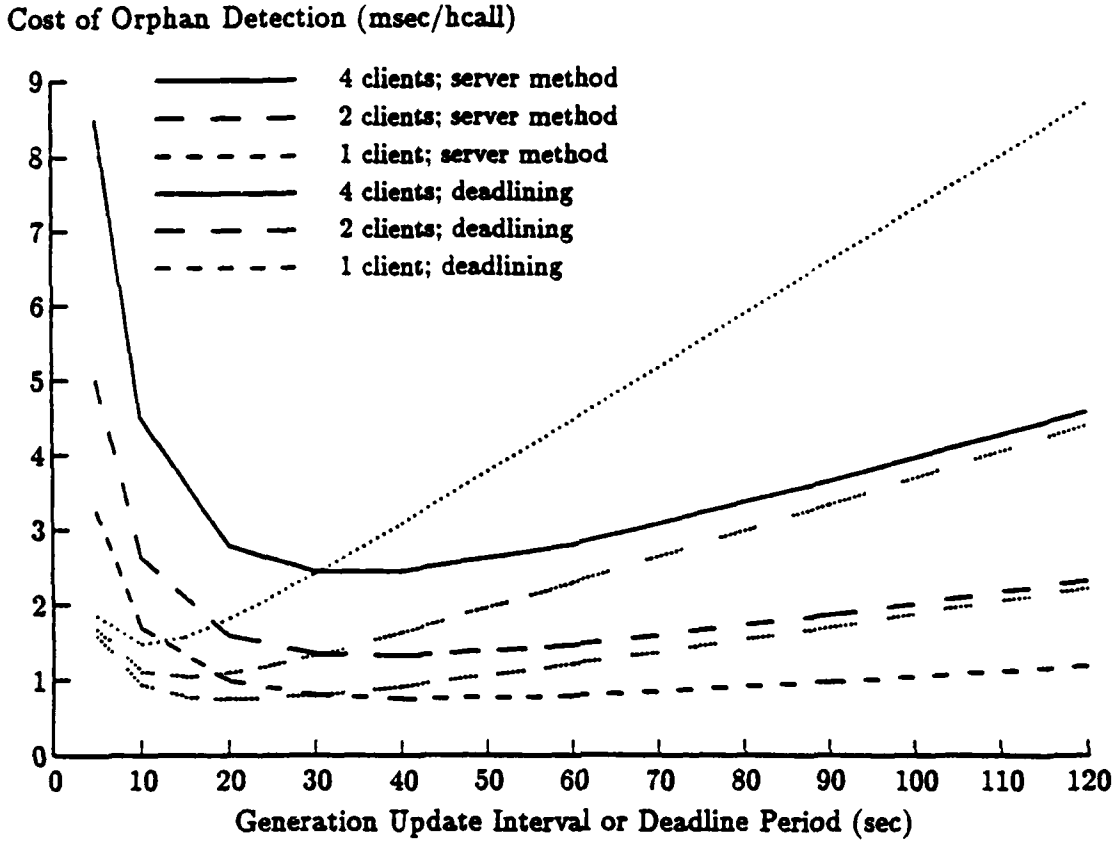


Figure 7.23: Cost of orphan detection with extension of deadlines and generations

Figure 7.23 shows the cost of orphan detection in the presence of extension, for one, two and four clients. When there are only one or two clients, the minimum cost is nearly the same for both methods. (The deadlining algorithm has a slightly lower minimum cost.) With four clients, the minimum cost of the deadlining algorithm is significantly lower than the cost of the central server algorithm. In general, the minimum cost of deadlining is less than the cost of the central server algorithm, as long as $xc < 17\text{msec}(n)$. This formula was obtained by calculating the minimum cost for each algorithm (differentiate cost formula, set equal to zero, substitute back in), setting the minimum costs equal, and solving for xc.

For the central server algorithm, the ideal generation update interval ui^* and minimum

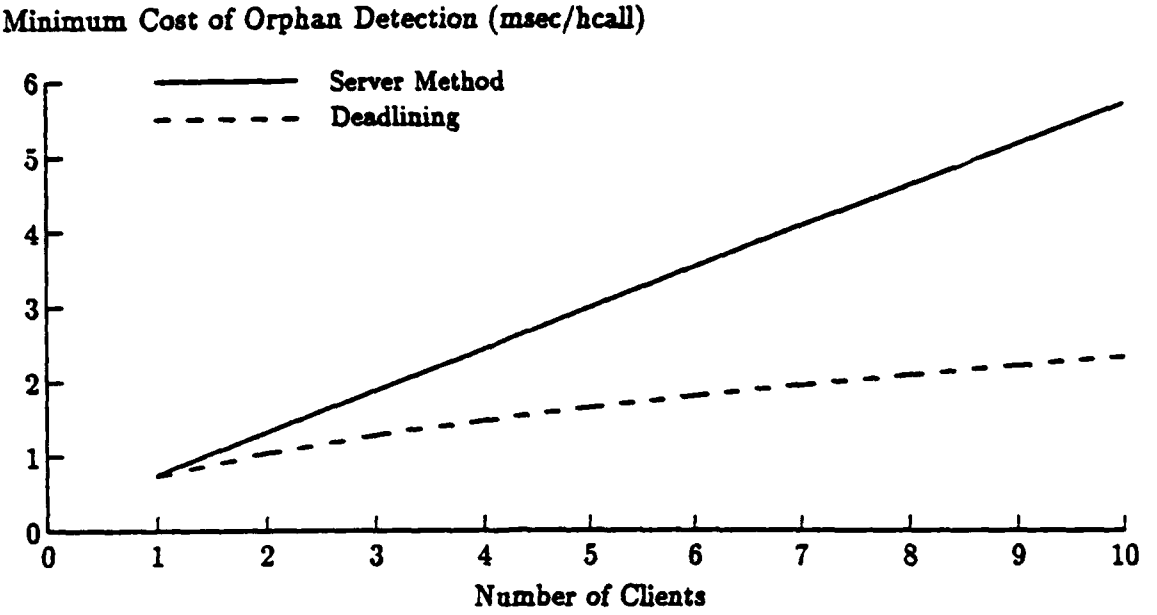


Figure 7.24: Cost of orphan detection vs. number of clients

cost c_{min} are given by the following formula:

$$ui^* = \sqrt{\frac{17\text{msec}(n) + xc}{17.6\mu\text{sec}(n)}}; \quad c_{min} = \sqrt{\frac{17.6\mu\text{sec}(n)}{17\text{msec}(n) + xc}}$$

The ideal deadline period dp^* and minimum cost for the deadlining scheme are as follows:

$$dp^* = \sqrt{\frac{xc}{36\mu\text{sec}(n)}}; \quad c_{min} = \sqrt{36\mu\text{sec}(n)(xc)}$$

Figure 7.24 shows the minimum orphan detection cost for the deadlining and central server algorithms, in the presence of extension. The cost of orphan detection was calculated for systems with one to ten clients.

7.6 Performance With a Large Number of Guardians

It is important to consider the behavior of the Argus orphan detection method in a system containing a large number of guardians (e.g. hundreds). Although we were unable to run experiments with so many guardians, we can still get some idea of the likely behavior of the algorithm.

If the central server algorithm is used for orphan detection in a large system, there may be problems caused by unnecessary *gmap* refreshes. Consider a large system with many guardians (hundreds, perhaps thousands), but which is divided into small, loosely coupled subsystems, with only occasionaly inter-subsystem communication. For example, suppose that guardians G_1 and G_2 communicate with each other most of the time, and rarely (or never) communicate with other guardians. Also, suppose that G_1 and G_2 initially have timestamp t_0. When G_1 notifies the generation service of a change in its generation count, a new *gmap* and a timestamp t_1 are returned. Subsequently, G_1 can make a handler call to G_2, with t_1 piggybacked on the call message. Because $t_1 > t_0$, G_2 must obtain a new *gmap* from the service before processing the handler call. However, if the number of guardians in the system is sufficiently large, there is a high probability that at least one guardian notified the server of a generation change after G_1 received the service's reply. In this case, the *gmap* returned to G_2 will reflect this change, if the update and *refresh* took place at the same replica, or if gossip has propagated between the two replicas. For a very large system, there would be a high probability of an update occuring at each replica, guaranteeing that the timestamp t_2 returned to G_2 will be larger than t_1. When G_1 receives G_2's reply, G_1 will need a new *gmap*, because the reply message's stamp t_2 is greater than G_1's stamp t_1. But when G_1 performs a *refresh*, there is a high probability that another guardian will have advanced its generation, causing the returned timestamp to be even larger than t_2. Thus, G_1 and G_2 are forced to get a new *gmap* from the server every time they receive a handler call or reply. Clearly, this is unacceptable behavior; it is as inefficient as sending the whole *gmap* in each call and reply. In fact it is worse, because of the time needed to make the extra calls to the server.

The above problem is unlikely to affect the crash orphan part of the algorithm. If guardian crashes are relatively rare, *map* will not change very often, so *map* updates will be much less frequent than *gmap* updates. Therefore, it is unlikely that the *map* will change so often as to force guardians to call the *map_service* each time a message is received.

The effects of system size on the size of *done* are similar for both the central server

scheme and the deadlining method. In a large, tightly coupled system, *done* may grow very large when either method is used. A tightly coupled system may result from a pattern of communication where every guardian talks directly to all others, or where all guardians are clients of shared services. The latter pattern is realistic for many applications. In this case, *done* entries will propagate rapidly among guardians. The *aid* of an action that aborts anywhere in the system will have a chance to propagate to all guardians before it is deleted. Thus, the size of *done* is proportional to the size of the whole system.

When either optimization is used in a large, loosely coupled system, the size of *done* should remain limited. We can define an "effective size" of the system, reflecting the number of other guardians with which a given guardian communicates, directly or indirectly. The "effective size" of a loosely coupled system should be defined in such a way that the average size of *done* for a system of given effective size k is the same as for a closed, tightly coupled system with k guardians. It is not clear how to determine the "effective size" of a system for a given pattern of communication; *this is an area for future research.*

If a system contains a group of m guardians that only communicate among themselves, we can take m to be the "effective size" of the system, as viewed by each of those guardians. The guardians in the group never receive *done* information from guardians outside the group, so the size of their copies of *done* should be proportional to m, not to the total system size n. If inter-subsystem communication does occur, but only rarely, many entries in *done* will be deleted before they are sent outside the local subsystem. When the deadlining method is used, the *aids* are deleted when they reach their deadlines; in the central server version of the algorithm, an *aid* is deleted when the *gmap* value of its origin is large enough. Thus the "effective size" of the system may be considerably less than the total number of guardians it contains.

Because the deadlining optimization treats *map* and *done* similarly, the above discussion of the size of *done* in loosely and tightly coupled systems should also apply to the size of *map*. However, the analysis is not very relevant for *map*, because deadlining should keep

the *map* small for realistic system sizes and map deadline periods. Also, map deadlines will probably be set long enough that most systems appear "tightly coupled" in the sense that *map* information will propagate to most guardians before being deleted.

Chapter 8

Conclusion

In this thesis, we presented an implementation of the central server optimization of the Argus orphan detection algorithm. We explained how the use of a central service to hold orphan detection information can reduce the amount of orphan information sent in normal system messages, thereby lowering the cost of the algorithm. Specifications for the central service were given, and its implementation was discussed in detail. We also described the modification of the Argus system to make use of the service for orphan detection. The effect of the central server algorithm on the cost of orphan-related operations and on overall system performance was measured under a variety of conditions. The central server scheme was also compared to another optimized version of the orphan detection algorithm, called *deadlining*.

8.1 Performance of Orphan Detection

The results in Chapter 7 show that the central server optimization is more practical than the unoptimized version of the Argus orphan detection algorithm. Instead of allowing *done* to grow without bound, the central server scheme keeps it at a relatively small size. However, even for small systems, the central server algorithm is somewhat inefficient. In a client-server system with four clients, the cost of orphan detection is three milliseconds per handler call, or approximately one-sixth of the time needed for a null handler call and reply.

The deadlining optimization is more efficient than the server method, for the systems that we have considered, provided that the deadline period is chosen appropriately. If actions are relatively short, a deadline period can be chosen that is long enough to make deadline extension unnecssary, but short enough to allow better performance than the server method. Even if actions take a long time to complete, making deadline extension and generation extension necessary, the minimum cost of orphan detection is smaller for the deadlining scheme.

Even though deadlining is more efficient than the server method, it is still somewhat costly. In a system with four clients and one server, the cost of orphan detection is 1–2 msec for a deadline period of 10-20 seconds. (If the deadline period is much shorter than this, we must take into consideration the overhead of identifying and deleting obselete *done* entries; also, it is more likely that deadline extension will be required.) If actions take a long time to complete, so that deadline extension is required, the minimum overhead is 1.5 msec per call. This is still a significant fraction of the time needed for a handler call.

It should be noted that the above results depend on our assumptions about the rate at which handler calls are made and the frequency with which they are aborted. If calls are made less often or if abort rates are less than in our experiments, the size of *done* and the overhead of orphan detection will be smaller.

The overhead of orphan detection can be reduced if we avoid adding entries to *done* unnecessarily. Before adding the *aid* for an aborted handler call to *done*, the Argus system can try to notify the target of the call that the call is being aborted. If the target guardian is contacted successfully, and is able to abort the call and destroy all of its descendants (by notifying other guardians about the abort, if necessary), then no orphans will be created by the abort. In this case, an entry for the call need not be added to *done*. Only if the system fails to locate and destroy the call's descendants must a *done* entry be added. It should be possible to eliminate the descendants of an aborted call in this manner except when some type of persistent failure has occurred, such as a guardian crash or network partition.

Therefore, the number of *crucial aborts* (i.e. aborts that cause entries to be added to *done*) should be considerably less than the total number of aborts. Because crucial aborts happen only in the presence of failures, we expect that their frequency will be much lower than the abort rates used in our experiments.

Another way to reduce the overhead of the central server algorithm is to reduce the frequency with which guardians communicate with the generation service. There are at least two ways to do this. One method is for guardians to notify the service of a generation change only when *done* contains entries that can be removed; the other involves passing some *gmap* information in messages between guardians. These methods can be used together.

Guardians should not notify the service of a generation change unless *done* contains entries from generations that have not yet been entered into the service. A guardian can check for this by examining its own copy of *done*. However, it is possible that a descendant of a topaction caused entries to be added to some other guardian's *done* and that these entries have not yet propagated to the topaction's guardian. This is possible only for active topactions; if *done* entries are created for a topaction's descendants, at least one is always added to the topaction's guardians's *done* by the time the topaction completes. We have not decided how to handle this case; one possibility is to query the guardians of descendants of active topactions, when necessary, to determine whether the descendants added entries to *done*.

In [6], it is proposed that guardians should keep track of recent changes to the generation map, and piggyback these changes in normal messages between guardians. A list of recent changes would be returned by the generation service, in addition to the new map and stamp, whenever an update or refresh operation was peformed. Each message between guardians would include a pair of timestamps t_1 and t_2, and a set of *gmap* entries. The meaning of this information would be as follows: t_2 would be the sender's timestamp, t_1 would be an earlier stamp, and the list of *gmap* entries would consist of those changes occuring "between" t_1 and t_2. In other words, if the recipient's timestamp is greater than or equal to t_1, the

recipient can merge these recent changes with its own *gmap* and merge t_2 with its own stamp, to produce a consistent and sufficiently recent state and stamp, without getting a new *gmap* from the generation service. The detailed implementation of this scheme remains to be developed.

In a large system, it appears that the deadlining method should give better performance than the server algorithm, provided that the system is loosely coupled. In this case, the cost of deadlining should depend on the "effective size" of the system, as seen by a guardian within it. Many *done* entries should be deleted because their deadlines have expired, before they have propagated to all of the other guardians. The cost of the server method, on the other hand, depends on the total size of the system. As explained in Section 7.6, when a guardian G notifies the generation service of a generation change, all of the other guardians may be forced to perform a refresh, even if they do not communicate with G.

If guardians exchange information about recent *gmap* changes, as suggested above, then the problem of guardians making too many calls to the generation service in large systems may be eliminated. When a guardian updated its generation it could include sufficient information in its handler calls to make it unnecessary for the recipient to contact the generation service. Thus, the recipient would not learn of additional generation updates by other guardians, and the sender would not be forced to call the service again when the call returned. Experiments are required to determine whether this optimization would make it practical to use the central server algorithm for systems with a large number of guardians.

One can imagine a large, tightly coupled system, in which *done* information propagates rapidly among guardians, and reaches most guardians before being deleted. For such a system, even the deadlining method may be too inefficient. In this case, *done* contains entries for aborts occurring anywhere in the system; it may, therefore, become excessively large. However, it is not clear that real systems will be this tightly coupled, especially if a short deadline period is used.

8.2 Other Orphan Detection Methods

Many other methods [3, 10, 11] of dealing with orphans have been proposed. Most of these involve preventing orphans from being created, rather than destroying them once they have been formed. In these algorithms, an action is not permitted to abort until all of its active descendants have been destroyed. Because of the possibility of communication failures, the time required to find and destroy an action's remote descendants is unbounded. Therefore, deadlines are used to limit the time an action must wait before aborting. An action's descendants are destroyed when their deadlines expire; this allows the parent action to abort once the deadline has passed.

It may be inconvenient to delay the abort of an action until its descendants have been destroyed. For example, if an action makes a request to one replica of a distributed service, and is then unable to communicate with the node at which the request is running, it may be desirable to abort the request immediately, and attempt to perform it at another site. This is not possible if the request cannot be aborted until the remote child action has been dest oyed. The use of deadlines limits the amount by which the abort is delayed; however, it may be preferable to abort immediately.

If deadlines are used to limit the delay when an action aborts, then it is possible that some remote calls will need to run beyond their deadlines. This makes it necessary to use some form of deadline extension, as in the deadlining optimization of the Argus algorithm. Deadlines must be chosen carefully, to avoid both excessively frequent extension and long delays when an action aborts.

It is difficult to compare the performance of this type of orphan elimination mechanism with the orphan detection algorithm used in Argus. The Argus algorithm produces a fairly constant orphan detection overhead for each handler call. This is especially true for the deadlining optimization; the central server optimization produces extra overhead when it is necessary to get a new *gmap* from the generation service. In contrast, the other methods

introduce delays when an action is aborted. When comparing the Argus algorithm with the others, it is not sufficient to determine the total length of all orphan-related delays; it is also necessary to decide the relative importance of delaying action aborts vs. adding overhead to each call.

8.3 Future Research

There remain several areas for future research concerning our orphan detection algorithm. One of these is determing how often crucial aborts occur in real systems. Because the performance of our algorithm depends heavily on the abort rate, it is useful to obtain realistic values for the frequency of crucial aborts.

Another area for research is the actual behavior of the algorithm in large systems. In section 7.6 we argued that in large systems, even if loosely coupled, it may be necessary for guardians to obtain new *gmaps* from the generation service on most calls. When a guardian G_1 notifies the service of a generation change, a new state is returned by the generation service. The next time G_1 sends a message to another guardian G_2, G_2 is forced to to perform a *refresh* to obtain the new state. However, by the time the *refresh* is performed, another guardian may have notified the service of a generation change; in this case the state returned to G_2 by the service is even newer than G_1's. When G_2 next communicates with G_1, G_1 is forced to perform a *refresh*, and may obtain a state even newer than G_2's. Theoretically, this could go on indefinitely, causing a *refresh* operation to be performed on every call and reply. Although this problem can, theoretically, arise in large systems, it remains to be determined whether it actually occurs in practice.

A third research area is the further optimization of the central server algorithm. In section 8.1, a number of optimizations were proposed. It would be interesting to determine how much they can improve the performance of the algorithm, and whether other optimizations are possible.

One final question is how to determine the "effective size" of a system for the purposes of

the deadlining algorithm. It would be interesting to know whether the effective size is much smaller than the total size in most systems. This would indicate whether the deadlining optimization is practical for systems with many guardians.

Appendix A

Calculating the Cost of Orphan Detection

Here we present our calculation of the total cost of orphan detection for the deadlining method and central server algorithm. These calculations are for a system consisting of a single server, a small number of clients and a one percent abort rate for handler calls. Each client starts a topaction every 500 msec and makes one handler call to the server per topaction. Only the cost of the abort orphan part of the algorithm is considered.

A.1 Cost of Deadlining

For the deadlining method, the principal cost of orphan detection is the time needed to piggyback *done* in messages, and to merge two copies of *done*. The cost per handler call can be calculated by multiplying the *done* processing time per message by the number of messages per handler call.

The *done* processing cost for a message is equal to the size of *done* multiplied by the processing time per entry. As shown in Section 7.4, when the deadlining method is used, the average size of *done* is $(4/230)(n)(dp)$, where n is the number of clients, and dp is the deadlining period in seconds. In section 7.3, we determined that the time needed to piggyback *done* on a message can be described by the following formula:

$$\text{time} = 15\text{msec} + \text{size-of-done} * (240\mu\text{sec} + \text{aid-size} * 40\mu\text{sec})$$

In the example we are considering, the size of each *aid* in *done* is three elements. There is

one element for the call's topaction ancestor, one for the call action and one for the handler action. Therefore, the cost of piggybacking *done* is 360μsec per entry.

The cost per handler call is three times this amount, because the number of messages with a piggybacked *done* is three times the number of handler calls. Each handler call requires two messages containing a copy of *done*: the call message, and the reply message. A third message containing *done* (the *prepare* message) is sent during two phase commit for the call's topaction. (Other messages used in two phase commit do not contain a copy of *done*.) If each topaction made several handler calls instead of just one, then the number of messages with piggybacked copies of *done* would be a little more than twice the number of handler calls. The cost of committing the topaction would be divided by the number of calls the topaction made, to obtain the cost per call.

In the deadlining scheme, the cost of merging two similar versions of *done* is 156μsec per entry, when each *aid* has three elements. Combining this with the 360μsec per entry cost of piggybacking gives a total cost of 516μsec per entry. Because there are three messages containing *done* for each handler call made, the total cost of *done* processing for each handler call is 1548μsec multiplied by the size of *done*. Using the calculated value for the size of *done*, we find that the total cost of processing *done* is 27μsec$(n)(dp)$.

A.2 Cost of Server Algorithm

For the central server algorithm, the total orphan detection cost is equal to the cost of processing *done* when messages are received plus the cost of processing *gmap*. The cost of processing *done* consists of the cost of piggybacking, and the cost of merging an incoming *done* with the guardian's own version. The *gmap* cost is composed of the time for which handler calls are delayed while a new *gmap* is obtained from a replica, the time needed to merge the new *gmap* with the guardian's own, and the processing time required to delete old entries from *done*, using the new *gmap* information.

For the small systems we are considering in this analysis, the time required to obtain

a new *gmap* from a replica is much larger than the time needed to process the new *gmap* after it has been received. In a system with 4 clients, *gmap* has 5 elements; in this case the combined cost of merging *gmap* and deleting old *done* entries is less than 10 percent of the cost of communicating with a replica of the service. Therefore, to simplify our analysis, we ignore the time needed to merge *gmap* and to remove old entries from *done*. We also ignore the time needed to do *gmap* processing for a generation update operation. The communication delay for updates is not included in our cost calculation because update operations do not delay handler calls; the guardian can still perform normal operations while waiting for the replica's response.

The cost of piggybacking *done* when the central server algorithm is used is 360μsec per entry, the same as for deadlining. In our implementation, the cost of merging two similar versions of *done* is 270μsec per entry, when each *aid* has three elements, assuming that *gmap* is small. Therefore the total cost is 630μsec per entry. Because there are three messages containing *done* for each handler call made, the total cost of piggybacking for each handler call is 1980μsec multiplied by the size of *done*.

As shown in Section 7.4, the size of *done* in a client-server system is approximately $.007n(ui)$, where n is the number of clients and ui is the generation update interval in seconds. The cost of processing *done* is therefore $13.2\mu\text{sec}(n)(ui)$ per handler call.

The cost of communicating with the generation service is calculated by multiplying the cost of each *refresh* operation by the ratio of refreshes to handler calls. All refreshes performed by the client guardian are included in this cost. Refreshes made by the server are included only if they delay a handler call made by a particular client.

The total time for which a particular client C's handler calls are delayed by communication with the generation service is equal to the delay per *refresh* operation multiplied by the number of refreshes. This time is divided by the number of handler calls made, to obtain the average delay per call. During a single generation update interval, each client notifies the service of a generation change exactly once. Each of these updates produces

one *refresh* operation that delays one of C's handler calls. When C performs an update, the server must get a new *gmap* to process C's next handler call. When any of the other $n-1$ clients performs an update, the server gets a new *gamp* to process that client's next call: C must then get a new *gmap* to process the next reply it receives. Therefore, C's calls are delayed n times per update interval. This means that the probability that a call is delayed is $n(ci/ui)$, where ci is the interval between calls. In our experiments, ci = 500msec. The results presented in Figure 7.11 confirm this estimate of the probability that a *refresh* is needed to process a call. The refresh probability for a client is close to $(n-1)$(500msec/ui). For a server the probability is close to (1/2)(500msec/ui) per message, or 500msec/ui per handler call.

For a small system, with a small *gmap*, the time needed to get a new *gmap* from a replica is 17 msec. Multiplying by the number of *refreshes* per handler call gives 8.5msec(n/ui) as the replica delay per handler call.

Adding together the cost of processing *done* and the cost of communicating with the generation service, gives the following formula for the total orphan detection overhead per handler call:

$$13.2\mu\text{sec}(n)(ui) + 8.5\text{msec}(n)/(ui)$$

A.3 Cost of Deadline and Generation Extension

We can also calculate the cost of orphan detection in the presence of long actions, when deadline extension and generation extension are necessary. We will consider the case of an single, long topaction that never completes. The topaction's handler calls are assumed to have a 500 msec service time. As soon as one call completes, the next call is made. As in most of our other examples, we choose a handler call abort rate of one percent.

The cost of deadline extension per handler call is equal to some fixed cost xc per extension, divided by the number of handler calls that run between extensions. We do not have data concerning the value of xc; in [12] information is provided about the frequency

of deadline extension, but not the time required. We will assume that the fixed cost of extension is equal to 15 msec, roughly the time needed for a low level send and reply. This seems reasonable, because extension involves changing one table entry and sending one short message to each of the topaction's active descendants; in our example, each topaction has only one active descendant at a time.

The total cost of orphan detection is obtained by adding the extension overhead to the cost in the absence of extension. The cost in the absence of extension is less than before, because we are no longer committing a topaction after each handler call. This means that only two messages containing a copy of *done*, not three, are sent for each call, and the *done* processing cost is only two thirds as large. The total orphan detection cost therefore, is

$$18\mu\text{sec}(n)(ui) + xc(ci/dp)$$

We must also consider the cost of extending actions' generations when the server algorithm is used. We have no data concerning the cost of generation extension, because extension of generations was not implemented for this thesis. It seems reasonable to use the same estimate for the cost extending deadlines and generations. In both cases, the amount of processing performed is similar. Like deadline extension, generation extension is performed once per update interval, in our example with a single, long topaction. (In some other cases the extension frequencies would not be the same.) This gives the following estimate for the cost of orphan detection in the presence of generation extension:

$$8.8\mu\text{sec}(n)(ui) + (17\text{msec}(n) + xc)/(2ui)$$

References

[1] Gray, J. "Notes on Database Operating Systems" in Bayer, R., ed., *Lecture Notes in Computer Science,* Vol. 60: *Operating Systems: An Advanced Course.* pp. 393–481. Springer-Verlag, 1981.

[2] Hwang, Deborah. *Constructing a Highly Available Location Service for a Distributed Environment.* M.I.T. L.C.S. Technical Report 410, January 1988.

[3] Lampson, Butler. "Applications and Protocols" In B. Lampson, ed., *Lecture Notes in Computer Science,* Vol. 105: *Distributed Systems: Architecture and Implementation: An Advanced Course.* Chapter 14. Springer-Verlag, 1981.

[4] Lampson, B. and Sturgis, H. *Crash Recovery in a Distributed Data Storage System.* Xerox Research Center, Palo Alto, California, 1979.

[5] Liskov, Barbara. *Overview of the Argus Language and System.* M.I.T. L.C.S. Programming Methodology Group Memo 40, February, 1984.

[6] Liskov, Barbara. *Highly Available Distributed Services.* M.I.T. L.C.S. Programming Methodology Group Memo 52, February 1987.

[7] Liskov, B., Curtis, D., Johnson, P., and Scheifler, R. "Implementation of Argus" In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles.* ACM Press, November, 1987.

[8] Liskov, B., Day, M., Herlihy, M., Johnson, P., Leavens, G., Scheifler R., and Weihl, W. *Argus Reference Manual.* M.I.T. L.C.S. Technical Report 400, November, 1987.

[9] Liskov, B., Scheifler, R., Walker, E. and Weihl, W. *Orphan Detection.* M.I.T. L.C.S. Programming Methodology Group Memo 53, February 1987.

[10] McKendry, M. and Herlihy, M. "Time Driven Orphan Elimination" In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems.* pp. 42–48. IEEE, January, 1986.

[11] Nelson, Bruce. *Remote Procedure Call.* Technical Report CMU-CS-81-119, Carnegie-Mellon University, 1981.

[12] Nguyen, Thu. *Performance Measurement of Orphan Detection in the Argus System.* Master's Thesis, June, 1988.

[13] Nguyen, Thu. Private Communication.

[14] Walker, Edward F. *Orphan Detection in the Argus System.* M.I.T. L.C.S. Technical Report 326, June, 1984.